Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

# Vysvetľovanie chýb vo formalizačných cvičeniach v logike prvého rádu

Bakalárska práca

2023 Karin Kubinová

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

# Vysvetľovanie chýb vo formalizačných cvičeniach v logike prvého rádu

Bakalárska práca

| | |
|---|---|
| Študijný program: | Aplikovaná informatika |
| Študijný odbor: | Informatika |
| Školiace pracovisko: | Katedra aplikovanej informatiky |
| Školiteľ: | Mgr. Ján Kľuka, PhD. |

Bratislava, 2023 Karin Kubinová


Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

# ZADANIE ZÁVEREČNEJ PRÁCE

**Meno a priezvisko študenta:** Karin Kubinová
**Študijný program:** aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
**Študijný odbor:** informatika
**Typ záverečnej práce:** bakalárska
**Jazyk záverečnej práce:** slovenský
**Sekundárny jazyk:** anglický

**Názov:** Vysvetľovanie chýb vo formalizačných cvičeniach v logike prvého rádu
*Explanation of errors in formalisation exercises in first-order logic*

**Anotácia:** Formalizácia patrí k najnáročnejším témam pri výučbe logiky. Jej cieľom je čo najpresnejšie zachytiť význam neformálneho tvrdenia v prirodzenom jazyku pomocou obmedzených prostriedkov formálneho jazyka. Výučba formalizácie sa podobá výučbe cudzieho jazyka. Na jej zvládnutie je potrebné spraviť množstvo cvičení a dostať na ne spätnú väzbu. Ideálne je, keď v prípade nesprávneho či neadekvátneho riešenia spätná väzba objasní, prečo študentovo riešenie nie je vhodné – napríklad formou kontrapríkladu. Poskytovať adekvátnu a promptnú spätnú väzbu v rozsahu, aký by niektorí študenti potrebovali, často nie je v silách učiteľov.

Hoci vo všeobecnosti formalizácia nemá jednoznačné riešenie, cvičenia na túto tému sa spravidla volia tak, aby boli čo najjednoznačnejšie a kombinovali známe štandardné idiómy. Navyše sú zvyčajne výsledkom formalizácie relatívne jednoduché formuly. Relatívna jednoduchosť a jednoznačnosť riešení poskytuje príležitosť využiť na kontrolu a poskytovanie spätnej väzby existujúce dokazovače pre logiku prvého rádu. V minulosti sme už takýto systém vyvinuli, ale spätnú väzbu, ktorú poskytuje, významná časť študentov nevyhodnotila ako nápomocnú. V práci na túto tému by sme preskúmať a otestovať pri výučbe ďalšie možnosti poskytovania vysvetlení chýb.

**Cieľ:** Navrhnúť a implementovať do existujúcej aplikácie na kontrolu formalizačných cvičení:
• zoskupovanie chybných riešení na základe ekvivalencie alebo ďalších relácií podobnosti medzi nimi navzájom a s očakávaným riešením;
• niekoľko techník na vysvetľovanie chýb a poskytovanie návodov na ich opravu;
• používateľské rozhranie pre učiteľov na zadávanie vstupov pre vysvetľovanie chýb a návody na opravu;
• používateľské rozhranie alebo iné mechanizmy na získavanie spätnej väzby od študentov na užitočnosť vysvetlení a návodov.
Vyhodnotiť užitočnosť vysvetlení a návodov vo výučbe.

**Literatúra:** Barker-Plummer, D., Barwise, J., Etchemendy, J. et al.: Language, Proof and Logic. Second Edition. Stanford: CSLI, 2010.


Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Barker-Plummer, D., Dale, R., Cox, R., Etchemendy, J. Automated Assessment in the Internet Classroom. In: Proceedings of the AAAI Fall Symposium on Education Informatics, Arlington, VA. AAAI 2008.
Perikos, I., Grivokostopoulou, F., Hatzilygeroudis, I. Automatic marking of NL to FOL conversions. In: Uskov, V. (ed.). Procs. Computers and Advanced Technology in Education (CATE 2012). ACTA Press, 2012.
Kovacs, L., Voronkov, A. First-Order Theorem Proving and Vampire. In: CAV 2013, LNCS 8044, Springer, 2013.
Alonso, J.-A., Aranda-Corral, G. A., Martín-Mateos, F. J.: KRRT: Knowledge Representation and Reasoning Tutor System. In: EUROCAST 2007, LNCS 4739, Springer 2007.
Hoder, K., Kovács, L., Voronkov, A.: Interpolation and Symbol Elimination in Vampire. In: IJCAR 2010, LNCS 6173, Springer 2010.

**Kľúčové slová:** nástroje pre logiku, logika prvého rádu, formalizácia, vysvetlenie, vyhodnocovanie cvičení, webová aplikácia

**Vedúci:** Mgr. Ján Kľuka, PhD.
**Katedra:** FMFI.KAI - Katedra aplikovanej informatiky
**Vedúci katedry:** doc. RNDr. Tatiana Jajcayová, PhD.

**Dátum zadania:** 01.09.2020

**Dátum schválenia:** 19.09.2022 doc. RNDr. Damas Gruska, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

**Poďakovanie:** Ďakujem môjmu školiteľovi Mgr. Jánovi Kľukovi, PhD. za cenné rady, ochotu a odborné vedenie počas písania tejto práce.

# Abstrakt

V tejto práci sa venujeme návrhu a implementácii nového spôsobu poskytovania spätnej väzby na chyby vo formalizačných cvičeniach v existujúcej webovej aplikácii, slúžiacej na ich kontrolu. Chybné riešenia tvrdení zoskupujeme na základe ekvivalencie. Učitelia si môžu zobraziť tieto zoskupené chybné formalizácie a k jednotlivým reprezentantom môžu pridať nápovede, ktoré slúžia na vysvetlenie chýb. Študenti si môžu, v prípade, že urobia chybnú formalizáciu ekvivalentnú niektorej, ktorá obsahuje pridané nápovede, postupne zobrazovať nápovede, pričom majú možnosť každú nápoveď ohodnotiť, podľa toho, či ju považujú za užitočnú. V predošlej implementácii sa študentovi, v prípade nesprávneho riešenia, poskytol kontrapríklad, pomocou výpisu prvorádovej štruktúry. Táto forma spätnej väzby však v mnohých prípadoch nebola postačujúca, alebo bola príliš zložitá na to, aby z nej študenti vedeli vyčítať svoje chyby. Aplikácia bola nasadená a otestovaná študentmi, pričom sa pri testovaní dosiahli pozitívne výsledky. Ukázalo sa, že študenti majú záujem zobrazovať si jednotlivé nápovede, ktoré v mnohých prípadoch taktiež viedli k úspešnému sformalizovaniu tvrdenia.

**Kľúčové slová:** nástroje pre logiku, logika prvého rádu, formalizácia, vysvetlenie, vyhodnocovanie cvičení, webová aplikácia

# Abstract

In this work, we design and implement a new way of providing feedback on errors in formalization exercises in an existing web application used to check their correctness. We group the incorrect proposition solutions on the base of equivalence. Teachers can view these grouped incorrect formalizations and can add hints to the individual representations to explain the errors. If students make an incorrect formalization equivalent to one that contains added hints, they can view the hints in sequence, with the option to rate each hint according to whether they find it useful. In the previous implementation, in the case of an incorrect solution, the student was provided with a counterexample, using a first-order structure. However, this form of feedback was in many cases insufficient or too complex for students to be able to read their errors from it. The application was deployed and tested by students, which resulted in a positive feedback. It showed that students were interested in viewing the individual hints, which in many cases also led to successful formalization of the proposition.

**Keywords:** logic tools, first-order logic, formalization, explanation, exercise evaluation, web application

# Obsah

**Úvod** **1**

**1 Logika prvého rádu** **3**
1.1 Syntax logiky prvého rádu . . . . . 3
1.2 Sémantika logiky prvého rádu . . . . . 5
1.3 Ekvivalencia formúl . . . . . 6

**2 Predchádzajúce práce** **9**
2.1 Práca S. Gombárovej . . . . . 9
2.2 Práca N. Kulíkovej . . . . . 10
2.3 Doterajšia implementácia . . . . . 11
2.4 Ukážka aplikácie . . . . . 13

**3 Použité technológie** **19**
3.1 Express . . . . . 19
3.2 React . . . . . 22
3.3 Redux a RTK Query . . . . . 23
3.4 Vampire . . . . . 26

**4 Požiadavky a návrh rozšírenia aplikácie** **27**
4.1 Požiadavky na aplikáciu . . . . . 27
4.2 Návrh aplikácie . . . . . 28
4.2.1 Zoskupovanie chybných riešení . . . . . 28
4.2.2 Pridávanie nápovedí . . . . . 29
4.2.3 Zobrazovanie a hodnotenie nápovedí . . . . . 29
4.2.4 Zobrazovanie postupu vlastných riešení študentovi . . . . . 29
4.2.5 Refaktorovanie predošlého kódu . . . . . 29
4.2.6 Úprava databázového modelu . . . . . 31

**5 Implementácia** **33**
5.1 Refaktorovanie predošlého kódu . . . . . 33
5.2 Zoskupovanie chybných riešení . . . . . 33

5.3 Pridávanie nápovedí . . . . . 34
5.4 Zobrazovanie a hodnotenie nápovedí . . . . . 35
5.5 Zobrazovanie postupu riešenia cvičení študentovi . . . . . 37
5.6 Zobrazovanie postupu študentov v riešení cvičení . . . . . 38

**6 Testovanie** **41**
6.1 Cieľ testovania . . . . . 41
6.2 Priebeh testovania . . . . . 41
6.3 Výsledok testovania . . . . . 42

**Záver** **45**

**Príloha A: Elektronická príloha** **49**

# Úvod

Matematika je plná dôkazov a tvrdení, no overiť ich pravdivosť v prirodzenom jazyku je často náročné. Na tieto účely sú najvhodnejšie formálne jazyky, pričom na preklad do nich je potrebná formalizácia. Správne sformalizovanie výrokov, je preto veľmi dôležité, no pre študentov zvyčajne aj veľmi náročné. Aby sa študentom uľahčilo osvojiť si pravidlá formalizácie, bola vytvorená webová aplikácia [1], neskôr doplnená o viacero funkcií [2], využívaná aj na hodinách predmetu Logika pre informatikov. Táto aplikácia umožňuje učiteľom pridávať a editovať rôzne formalizačné cvičenia, zobrazovať postup študentov v riešeniach zadaní. Študentom je umožnené precvičiť si formalizáciu rôznych tvrdení, pričom sa automaticky kontroluje správnosť ich riešení. Pre jasnejšie pochopenie dôvodu, prečo nie je dané riešenie správne, sa študentovi vypíše štruktúra, v ktorej je študentova formalizácia nepravdivá, ale hľadaná správna formalizácia je pravdivá, prípadne naopak, štruktúra, v ktorej je študentova formalizácia pravdivá, ale hľadaná správna formalizácia je nepravdivá. Tento štýl spätnej väzby však v mnohých prípadoch študentom nepomohol jasne pochopiť, prečo je ich riešenie chybné.

V tejto práci sa preto venujeme inej možnosti, ako spätnú väzby poskytnúť. Chybné riešenia môžeme zoskupovať na základe ekvivalencie. Tento zoznam zoskupených chybných formalizácií ďalej zobrazíme učiteľom, ktorí budú môcť k jednotlivým reprezentantom chybných riešení pridať nápovede, opisujúce chyby v danom riešení. Tieto nápovede potom zobrazíme študentom, hneď ako urobia chybné riešenie ekvivalentné tomu, ktoré už má zadanú nejakú nápoveď od učiteľa. Študenti budú mať takisto možnosť ohodnotiť, či je pre nich daná nápoveď užitočná, alebo nie. Učitelia budú mať k dispozícii túto štatistiku, a aj na jej základe sa rozhodnúť, či sa bude daná nápoveď zobrazovať študentom.

V nasledujúcej kapitole 1 si zadefinujeme všetky dôležité pojmy k logike prvého rádu a ukážeme si aj zopár príkladov formalizácie. V kapitole 2 sa budeme venovať predošlým bakalárskym prácam, na ktoré nadväzuje táto práca. Takisto si vysvetlíme rôzne nedostatky, ktoré sa pokúsime vylepšiť v tejto práci. Rôzne technológie, ktoré využijeme si priblížime v kapitole 3. V kapitole 4 sa budeme venovať požiadavkám na naše rozšírenie aplikácie, ako aj návrhu ich implementácie. Samotnú implementáciu jednotlivých požiadaviek ďalej popíšeme v kapitole 5. V kapitole 6 si predstavíme výsledky z testovania aplikácie.

# Kapitola 1

# Logika prvého rádu

V tejto kapitole si zosumarizujeme všetky základné definície a pojmy, patriace k jazyku logiky prvého rádu. Vychádzame pritom z definícií predstavených na prednáškach [3]. Formálny jazyk je zjednodušený model prirodzeného jazyka. V matematike sa využíva vďaka tomu, že jeho syntax a sémantika sú jednoznačne definované. Nemôže sa teda stať, že by boli slová, či vzťahy medzi slovami nejednoznačné, čo sa v prirodzenom jazyku stáva pomerne často.

*Logika prvého rádu*, označovaná aj predikátová logika, rozširuje výrokovú logiku, ktorá je schopná vyjadriť iba jednoduché vzťahy medzi výrokmi. Keď teda pomocou zadefinovania predikátov, funkcií a kvantifikátorov zovšeobecníme výrokovú logiku, získame predikátovú logiku, ktorá je schopná vyjadriť aj zložitejšie vzťahy [4].

## 1.1 Syntax logiky prvého rádu

Syntax logiky prvého rádu špecifikuje z čoho sa skladajú prvorádové výroky a popisuje ich štruktúru.

*Symboly* jazyka $\mathcal{L}$ logiky prvého rádu delíme na 4 podskupiny:

1. *Indivíduové premenné* predstavujúce objekty, ktoré nie sú jasne špecifikované, z nejakej nekonečnej spočítateľnej množiny $\mathcal{V}_\mathcal{L}$.

2. *Mimologické symboly* kam patria:

   - *indivíduové konštanty* označujúce konkrétne objekty, z nejakej spočítateľnej množiny $\mathcal{C}_\mathcal{L}$,
   - *funkčné symboly* označujúce vzťahy medzi jednoznačne určenými objektami, z nejakej spočítateľnej množiny $\mathcal{F}_\mathcal{L}$,
   - *predikátové symboly* označujúce vzťahy a vlastnosti, z nejakej spočítateľnej množiny $\mathcal{P}_\mathcal{L}$.

3. *Logické symboly* kam patria:

   - *logické spojky*: unárna negácia $\neg$ a binárne: konjunkcia $\wedge$, disjunkcia $\vee$ a implikácia $\rightarrow$,
   - *symbol rovnosti* $\doteq$,
   - *kvantifikátory*: všeobecný $\forall$ a existenčný $\exists$.

4. *Pomocné symboly* (,) a , (ľavá, pravá zátvorka a čiarka).

Je nutné ešte špecifikovať, že množiny $\mathcal{V}_\mathcal{L}$, $\mathcal{C}_\mathcal{L}$, $\mathcal{F}_\mathcal{L}$ a $\mathcal{P}_\mathcal{L}$ navzájom neobsahujú žiadny spoločný prvok, a teda sú vzájomne disjunktné. Zároveň tieto množiny neobsahujú žiadny z logických, či pomocných symbolov.

Každý predikátový a funkčný symbol $S$ má vždy priradenú nejakú aritu $ar(S) \in \mathbb{N}^+$, ktorá určuje počet jeho argumentov.

Pre priblíženie prvorádového jazyka $\mathcal{L}$ uvedieme jednoduchý príklad:

$$\begin{gathered}\text{indivíduové premenné: } \mathcal{V}_\mathcal{L} = \{x, y, \ldots\} \\ \text{indivíduové konštanty: } \mathcal{C}_\mathcal{L} = \{\text{Karol, Janka}\} \\ \text{funkčné symboly: } \mathcal{F}_\mathcal{L} = \{\text{mama}^1\} \\ \text{predikátové symboly: } \mathcal{P}_\mathcal{L} = \{\text{súrodenec}^2\}\end{gathered}$$

*Termy* jazyka $\mathcal{L}$ logiky prvého rádu definujeme nasledovne.

Všetky indivíduové premenné a indivíduové konštanty jazyka $\mathcal{L}$ sú termy.

Ak $f$ je funkčný symbol jazyka $\mathcal{L}$ s aritou $n$ a zároveň $t_1, \ldots, t_n$ sú termy, tak aj $f(t_1, \ldots, t_n)$ je term.

Nič iné termom jazyka $\mathcal{L}$ nie je.

Vo vyššie uvedenom príklade sú termami, napríklad:

$$x,\ y,\ \text{Karol},\ \text{Janka},\ \text{mama(Karol)},\ \text{mama}(x),\ \ldots$$

*Rovnostný atóm* jazyka $\mathcal{L}$ je každá postupnosť symbolov $t_1 = t_2$, pričom $t_1$ a $t_2$ sú termy jazyka $\mathcal{L}$. Rovnostné atómy v predošlom príklade sú, napríklad:

$$x \doteq \text{Karol},\ \text{Janka} \doteq \text{mama}(x),\ \ldots$$

*Predikátový atóm* jazyka $\mathcal{L}$ je každá postupnosť symbolov $P(t_1, \ldots, t_n)$, pričom $P$ je predikátový symbol jazyka $\mathcal{L}$ s aritou $n$ a $t_1, \ldots, t_n$ sú termy jazyka $\mathcal{L}$. Predikátové atómy z príkladu sú, napríklad:

$$\text{súrodenec}(x, y),\ \text{súrodenec(Karol, mama(Janka))},\ \ldots$$

*Atomické formuly* alebo aj *atómy* jazyka $\mathcal{L}$ sú všetky rovnostné a predikátové atómy jazyka $\mathcal{L}$. Množinu týchto atómov označujeme $\mathcal{A}_\mathcal{L}$.

*Formuly* jazyka $\mathcal{L}$ logiky prvého rádu definujeme nasledovne.

Každý atóm z $\mathcal{A}_\mathcal{L}$ je formulou.

Ak je $A$ formulou jazyka $\mathcal{L}$, tak aj jej negácia $\neg A$ je formulou.

Ak sú $A$ a $B$ formulami jazyka $\mathcal{L}$, tak aj konjunkcia $(A \wedge B)$, disjunkcia $(A \vee B)$ a implikácia $(A \to B)$ sú formuly.

Ak $x$ je indivíduová premenná jazyka $\mathcal{L}$ a $A$ je formulou jazyka $\mathcal{L}$ , tak aj výrazy $\exists x\ A$ a $\forall x\ A$ sú formuly.

Nič iné formulou jazyka $\mathcal{L}$ nie je.

Formulami jazyka $\mathcal{L}$ sú napríklad:

$$(\text{súrodenec(Janka, Karol)} \wedge \neg \text{Karol} \doteq \text{Janka}),\ (\exists x\ (\text{mama}(x) \to x \doteq \text{Janka})),\ \ldots$$

## 1.2 Sémantika logiky prvého rádu

Na interpretáciu formúl využívame štruktúry, nazývané aj interpretácie jazyka. *Štruktúru* pre jazyk $\mathcal{L}$ značíme $\mathcal{M} = (D, i)$, pričom $D$ označuje *doménu* – ľubovoľnú neprázdnu množinu a $i$ predstavuje *interpretačnú funkciu* štruktúry $\mathcal{M}$ – zobrazenie, ktoré priraďuje :

- každej indivíduovej konštante $c \in \mathcal{C}_\mathcal{L}$ nejaký prvok $i(c) \in D$.
- každému funkčnému symbolu $f \in \mathcal{F}_\mathcal{L}$ s aritou $n$ funkciu $i(f) : D^n \to D$.
- každému predikátovému symbolu $P \in \mathcal{P}_\mathcal{L}$ s aritou $n$ množinu $i(P) \subseteq D^n$.

*Ohodnotením indivíduových premenných* nazývame ľubovoľnú funkciu, ktorá premenným priraďuje prvky domény. Značíme ju $e : \mathcal{V}_\mathcal{L} \to D$. Ďalej nech $x$ je indivíduová konštanta z $\mathcal{L}$ a $d$ je prvok z domény $D$. Zápisom $e(x/d)$ značíme ohodnotenie indivíduových premenných, ktoré premennej $x$ priraďuje hodnotu $d$ a všetkým ostatným premenným rovnakú hodnotu ako im priraďuje $e$, čo môžeme zapísať ako:

$$e(x/d)(y) = \begin{cases} d, & \text{ak } y = x \\ e(y), & \text{ak } y \neq x. \end{cases}$$

*Hodnotu termu* $t$ značíme $t^{\mathcal{M}}[e]$ a definujeme nasledovne, pre všetky premenné $x$, konštanty $a$ a funkčné symboly $f$ s aritou $n$:

$$x^{\mathcal{M}}[e] = e(x)$$
$$a^{\mathcal{M}}[e] = i(a)$$
$$(f(t_1, \ldots, t_n)^{\mathcal{M}}[e] = i(f)(t_1^{\mathcal{M}}[e], \ldots, t_n^{\mathcal{M}}[e])$$

Vzťah *štruktúra* $\mathcal{M}$ *spĺňa formulu* $X$ *pri ohodnotení* $e$ značíme $\mathcal{M} \models X[e]$ a definujeme induktívne nasledovne, pre všetky arity $n > 0$, predikátové symboly $P$ s aritou $n$, termy $t_1, \ldots, t_n$, premenné $x$ a formuly $A$ a $B$ jazyka $\mathcal{L}$:

- $\mathcal{M} \models t_1 \doteq t_2[e]$ vtt $t_1^{\mathcal{M}}[e] = t_2^{\mathcal{M}}[e]$,
- $\mathcal{M} \models P(t_1, \ldots, t_n)[e]$ vtt $(t_1^{\mathcal{M}}[e], \ldots, t_n^{\mathcal{M}}[e]) \in i(P)$,
- $\mathcal{M} \models \neg A[e]$ vtt $\mathcal{M} \not\models A[e]$,
- $\mathcal{M} \models (A \wedge B)[e]$ vtt $\mathcal{M} \models A[e]$ a súčasne $\mathcal{M} \models B[e]$,
- $\mathcal{M} \models (A \vee B)[e]$ vtt $\mathcal{M} \models A[e]$ alebo $\mathcal{M} \models B[e]$,
- $\mathcal{M} \models (A \to B)[e]$ vtt $\mathcal{M} \not\models A[e]$ alebo $\mathcal{M} \models B[e]$,
- $\mathcal{M} \models \exists x\, A[e]$ vtt pre nejaký prvok $d \in D$ máme $\mathcal{M} \models A[e(x/d)]$,
- $\mathcal{M} \models \forall x\, A[e]$ vtt pre každý prvok $d \in D$ máme $\mathcal{M} \models A[e(x/d)]$.

## 1.3 Ekvivalencia formúl

Formuly $A$ a $B$ sú *ekvivalentné* vtedy a len vtedy, keď pre každú štruktúru $\mathcal{M}$ a pre každé ohodnotenie $e$ platí $\mathcal{M} \models A[e]$ práve keď $\mathcal{M} \models B[e]$. Takýto vzťah ekvivalencie zapisujeme $A \Leftrightarrow B$.

Ekvivalencia formúl je pri formalizácii veľmi dôležitá vlastnosť, keďže tvrdenia vedia študenti vyjadriť viacerými ekvivalentnými formulami, a teda je dôležité uznať za správnu nie len nejakú jednu správnu formalizáciu, ale aj všetky k nej ekvivalentné.

Majme jazyk s takýmito predikátovými symbolmi a ich významami:

- `predmet(x)` – $x$ je predmet
- `vyberovy(x)` – $x$ je výberový
- `zapisany(x,y)` – $x$ má zapísaný $y$.

Ako príklad uveďme sformalizovanie tvrdenia:

Nikto si nezapisuje výberové predmety.

Toto tvrdenie vieme sformalizovať viacerými spôsobmi, napríklad:

$$\neg\exists x\ \exists y\ ((\texttt{predmet}(y) \wedge \texttt{vyberovy}(y)) \wedge \texttt{zapisany}(x, y)) \tag{1.1}$$

$$\forall x\ \forall y\ ((\texttt{predmet}(y) \wedge \texttt{vyberovy}(y)) \rightarrow \neg\texttt{zapisany}(x, y)) \tag{1.2}$$

$$\forall x\ ((\texttt{predmet}(x) \wedge \texttt{vyberovy}(x)) \rightarrow \neg\exists y\ (\texttt{zapisany}(y, x))) \tag{1.3}$$

Aj keď sa nám na prvý pohľad môže zdať, že všetky tieto formalizácie sú odlišné, v skutočnosti sú si navzájom ekvivalentné, a zároveň sú ekvivalentné správnej formalizácii, a teda sú všetky správne. Avšak tvrdenia, ktoré v prirodzenom jazyku obsahujú viaceré zápory, bývajú pre študentov obtiažnejšie na formalizáciu. Ukážeme si preto niekoľko chybných formalizácií tohto tvrdenia a vysvetlíme si chyby v nich.

$$\forall x\ \forall y\ ((\texttt{predmet}(x) \wedge \texttt{vyberovy}(x)) \wedge \neg\texttt{zapisany}(x, y)) \tag{1.4}$$

$$\neg\exists x\ \exists y\ ((\texttt{predmet}(x) \wedge \texttt{vyberovy}(x)) \wedge \neg\texttt{zapisany}(x, y)) \tag{1.5}$$

$$\neg\exists x\ \exists y\ ((\texttt{predmet}(y) \wedge \texttt{vyberovy}(y)) \rightarrow \neg\texttt{zapisany}(x, y)) \tag{1.6}$$

$$\forall x\ \exists y\ ((\texttt{predmet}(y) \wedge \texttt{vyberovy}(y)) \wedge \neg\texttt{zapisany}(x, y)) \tag{1.7}$$

Najčastejšou chybou je zviazanie všeobecného kvantifikátora s konjunkciou, čo je ukázané formalizáciou 1.4. Táto formalizácia, ak ju preložíme do prirodzeného jazyka, hovorí, že každý objekty z domény je predmet, výberový a nemá zapísaný žiadny ďalší objekt. Ďalšou chybou je neuvedomenie si toho, že dvojitá negácia z prirodzeného jazyka sa v logike zapisuje iba raz, ako môžeme vidieť na chybnej formalizácii 1.5. Študenti si v týchto formalizáciách,1.4 a 1.5, navyše zamieňajú poradie premenných v predikáte `zapisany`. Medzi ďalšie chybné formalizácie patria, napríklad formalizácie 1.6 a 1.7. Žiadne z formalizácií 1.4 až 1.7 si nie sú navzájom ekvivalentné, a rovnako nie sú ekvivalentné ani správnym formalizáciám 1.1 až 1.3. Aj napriek tomu, že formalizácie 1.4 a 1.5 patria medzi najčastejšie chybé riešenia, dokazovaču Vampire sa v týchto prípadoch nepodarilo nájsť žiaden rozumný kontrapríklad. Študenti pri týchto formalizáciách teda nemajú k dispozícií žiadnu spätnú väzbu, ktorá by im pomohla v riešení tvrdenia.

# Kapitola 2

# Predchádzajúce práce

V tejto kapitole si priblížime dve predchádzajúce bakalárske práce, na ktoré nadväzuje táto bakalárska práca. Prvou z nich je práca Samanthy Gombárovej, *Automatizácia kontroly formalizačných cvičení v logike prvého rádu* [1], ktorej výsledky boli neskôr rozšírené bakalárskou prácou Nikoly Kulíkovej, *Automatická spätná väzba na riešenia formalizačných cvičení* [2].

## 2.1 Práca S. Gombárovej

Cieľom práce S. Gombárovej bolo vytvoriť webovú aplikáciu, ktorá by zautomatizovala kontrolu formalizačných cvičení.

Pri tvorbe tejto aplikácie sa na strane klienta využili technológie React na tvorbu používateľského rozhrania a Redux na spravovanie stavu aplikácie. Na strane servera bol použitý framework Express, na komunikáciu medzi klientom a serverom a na prácu s databázou bol použitý databázový systém PostgreSQL.

Administrátorovi aplikácie je umožnené pridávať nové cvičenia, pričom je potrebné vyplniť názov cvičenia a tvrdenie, ktoré chceme aby používatelia sformalizovali. Každému tvrdeniu je potrebné zadať aspoň jednu správnu formalizáciu. K jednému cvičeniu je možné pridať viacero takýchto tvrdení na sformalizovanie. Po odoslaní sa cvičenie uloží do databázy.

Bežnému používateľovi sa zobrazuje zoznam cvičení. Po vybratí nejakého cvičenia sa zobrazia podrobnosti k cvičeniu, ako zoznam indivíduových konštánt, funkčné a predikátové symboly a tvrdenia na sformalizovanie. Pri každom tvrdení sa nachádza pole, kam môžu používatelia napísať svoje riešenie. Po kliknutí na tlačidlo sa na pozadí, dokazovačom Vampire, overí, či je daná formalizácia ekvivalentná správnemu riešeniu a naopak, či je správne riešenie ekvivalentné formalizácii používateľa. Používateľovi sa následne zobrazí správa, či je jeho riešenie správne, alebo nie.

Výsledná aplikácia umožňovala pridávať nové cvičenia a zobrazovať zoznam cvičení

s možnosťou ich riešiť. Používateľovi sa po odoslaní riešenia zobrazil oznam o správnosti riešenia. To však bolo, v prípade nesprávnej formalizácie, veľkým nedostatkom. Študentom by bolo vhodné naznačiť, kde by sa v ich riešení mohla nachádzať chyba. Takisto sa riešenia formalizácií nijak neukladali a tým pádom ich administrátori nemali možnosť analyzovať. Vyučujúci teda nemali prehľad o tom, ako sa študentom darí v riešení rôznych formalizácií. Do aplikácie sa mohol prihlásiť iba jeden používateľ – administrátor, ktorý mohol pridávať cvičenia bez možnosti ich editovať, či mazať. Cvičenia sa pritom mohli riešiť aj bez potreby prihlásenia.

## 2.2 Práca N. Kulíkovej

Cieľom práce N. Kulíkovej bolo rozšíriť spätnú väzbu, na riešenia študentov v predošlej práci, o generovanie konečných prvorádových štruktúr, a takisto umožniť učiteľom sledovať postup študentov v riešení cvičení.

Po odoslaní riešenia sa zavolá dokazovač Vampire, ktorý overí správnosť riešenia. Následne, v prípade nesprávneho riešenia, sa opäť zavolá Vampire, tentokrát aj s vhodnými prepínačmi, ktoré slúžia na získanie štruktúry. Vampire sa najskôr spustí tak, aby sa pokúsil nájsť štruktúru, v ktorej je študentova formalizácia pravdivá, ale správna formalizácia je nepravdivá. Následne sa Vampire spustí znova, pričom sa pokúsi nájsť štruktúru, v ktorej je študentova formalizácia nepravdivá, ale správna formalizácia je pravdivá.

Do aplikácie je možné sa prihlásiť aj pomocou portálu GitHub. Po stlačení príslušného tlačidla je používateľ presmerovaný na portál GitHub, kde sa musí autorizovať a povoliť tejto aplikácii prístup k jeho údajom. Následne je používateľ opäť presmerovaný na našu aplikáciu a po úspešnom prihlásení môže riešiť úlohy, pričom sa jeho postup riešení cvičení ukladá do databázy.

Učitelia môžu sledovať postup študentov v riešení úloh. V príslušnej karte sa učiteľovi zobrazí zoznam všetkých cvičení aj so štatistikou, koľko študentov sa pokúsilo vyriešiť danú úlohu. Po vybratí nejakej úlohy sa zobrazí zoznam študentov, ktorí sa ju pokúsili vyriešiť. Pri každom študentovi sa tiež zobrazuje štatistika, koľko tvrdení z danej úlohy sa pokúsil vyriešiť a koľko z nich aj správne vyriešil. Takisto sa zobrazuje aj celkový počet pokusov a počet úspešných pokusov, spolu s dátumom posledného pokusu riešenia. Po rozkliknutí nejakého študenta, vidí učiteľ samotné tvrdenia na formalizáciu a všetky študentove riešenia k nim, spolu s dátumom a správnosťou daného riešenia.

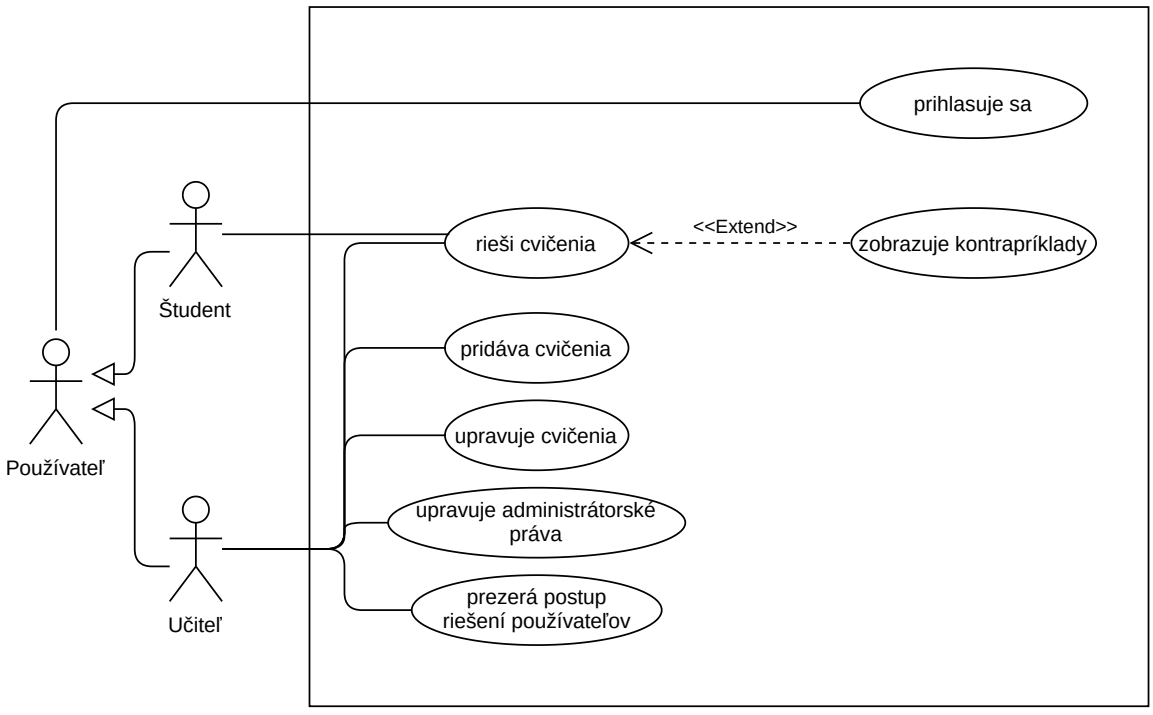
Učiteľ môže okrem pridávania nových cvičení navyše editovať už uložené cvičenia, prípadne ich zmazať. Takisto môže prideliť administrátorské práva aj iným používateľom, zvyčajne ďalším vyučujúcim.

Ani táto forma spätnej väzby pomocou kontrapríkladov však často nie je postačujúca. V niektorých prípadoch sú štruktúry príliš komplikované na to, aby z nich študenti vedeli vyčítať svoju chybu, alebo naopak sú príliš jednoduché na to, aby študentom dali relevantné informácie. Preto je potrebná ešte ďalšia forma spätnej väzby, aj vďaka ktorej študenti pochopia, kde sa nachádzajú chyby v ich riešeniach. Takisto je potrebné upraviť získavanie dát, na výpis štatistiky, pri zobrazovaní postupu študentov v riešení cvičení, pretože niektoré údaje nie sú správne, ako si ukážeme v sekcii 2.4.

## 2.3 Doterajšia implementácia

V tejto sekcii si priblížime doterajšiu implementáciu jednotlivých funkcií aplikácie a ukážeme si, ktoré časti by bolo potrebné upraviť.

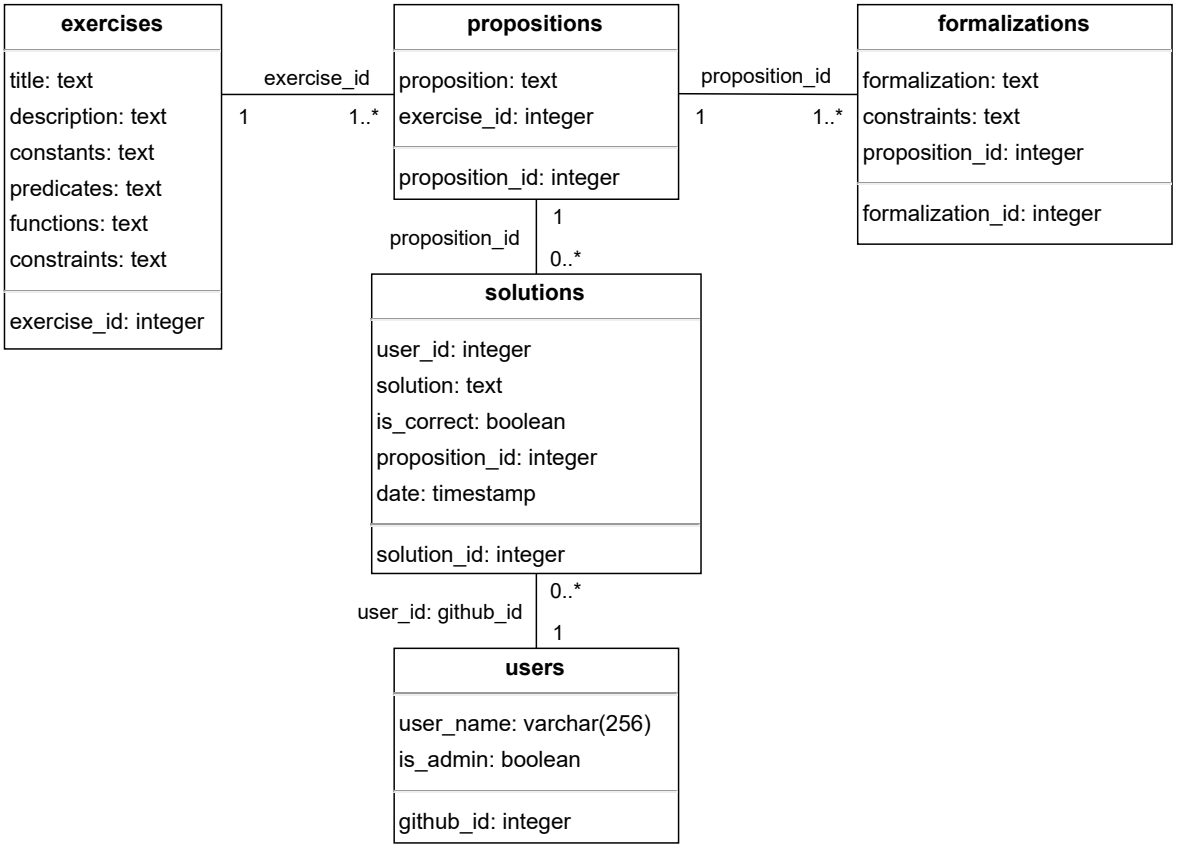
Aplikácia poskytuje rôznym používateľom rôzne funkcie. Po prihlásení môžu všetci používatelia zobrazovať a riešiť cvičenia. Ak má prihlásený používateľ administrátorské práva, zvyčajne v prípade učiteľov, má navyše možnosť pridávať a upravovať cvičenia, zobrazovať postup všetkých používateľov v riešení úloh a upravovať administrátorské práva iných používateľov. Tieto funkcie znázorňuje use case diagram na obrázku 2.1.




Obr. 2.1: Pôvodný use case diagram

Tieto rôzne funkcie do veľkej miery narábajú s databázou, či už na zobrazovanie dát, alebo na ukladanie nových dát. Aktuálna schéma databázy s jej tabuľkami je znázornená na obrázku 2.2. V tabuľke `exercises` sú uložené samotné cvičenia spolu

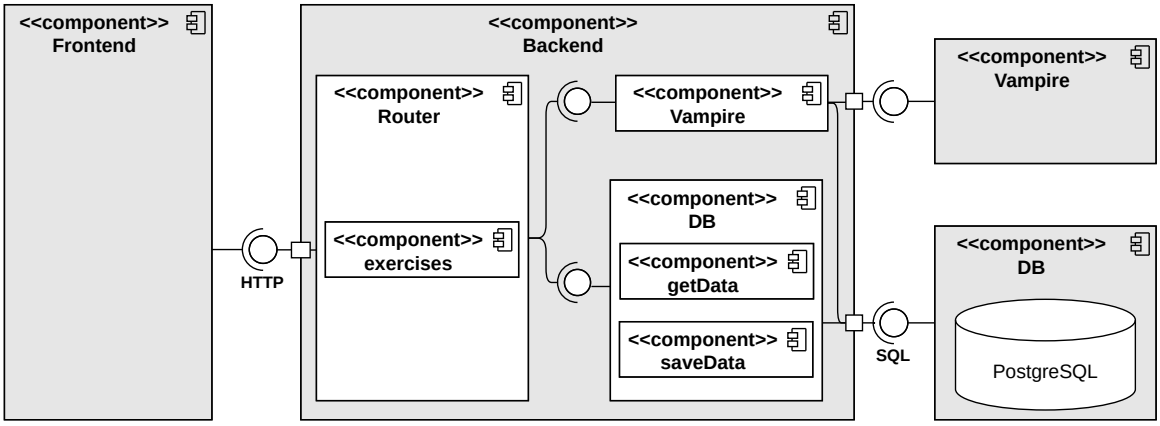
s jazykom daného cvičenia. Jednotlivé tvrdenia daného cvičenia sú uložené v tabuľke `propositions`. Každý záznam z tejto tabuľky obsahuje tvrdenie na formalizáciu a cudzí kľúč odkazujúci na id cvičenia, pomocou ktorého sa priraďujú jednotlivé tvrdenia k cvičeniam. Každé tvrdenie má aspoň jednu správnu formalizáciu. Tie sú uložené v tabuľke `formalizations` a okrem samotnej formalizácie obsahujú cudzí kľúč odkazujúci na id tvrdenia, ku ktorému patria. Jednotlivé riešenia tvrdení sa evidujú v tabuľke `solutions`. Každý záznam z tejto tabuľky obsahuje samotné riešenie tvrdenia, informáciu o tom, či je riešenie správne a dátum riešenia. Aby sme vedeli priradiť, k akému tvrdeniu patrí dané riešenie, evidujeme cudzí kľúč odkazujúci na id príslušného tvrdenia. Takisto evidujeme používateľa, ktorý je autorom daného riešenia, pomocou cudzieho kľúču odkazujúceho na jeho id z tabuľky `users`. V tejto tabuľke evidujeme používateľov, spolu s informáciou, či vlastnia administrátorské práva.




Obr. 2.2: Pôvodná schéma databázy

Avšak samotné členenie kódu na strane servera, ktorý narába s databázou, nie je navrhnuté ideálne. V pôvodnej implementácii sú moduly rozdelené vzhľadom na to, či dáta z databázy čítame, alebo ich zapisujeme. Takto rozdelený kód však nie je najprehľadnejší, a preto by bolo vhodné rozdeliť jednotlivé funkcie do modulov podľa typu dát, s ktorými narábajú. Taktiež by bolo vhodné oddeliť databázové transakcie od samotného dopytovania dát.

Aktuálne sa na strane servera využíva jeden modul `api/exercises.js`, ktorý obsluhuje požiadavky zo strany klienta na rôznych cestách. Tieto požiadavky sa však netýkajú iba práce s cvičeniami, ale napríklad aj prihlasovania používateľov, či získavania údajov na zobrazovanie postupu študentov v riešení cvičení. Bolo by teda potrebné kód viac zmodularizovať a oddeliť narábanie s rôznymi cestami do samostatných modulov. Aktuálna komunikácia jednotlivých komponentov aplikácie je znázornená komponentovým diagramom na obrázku 2.3.

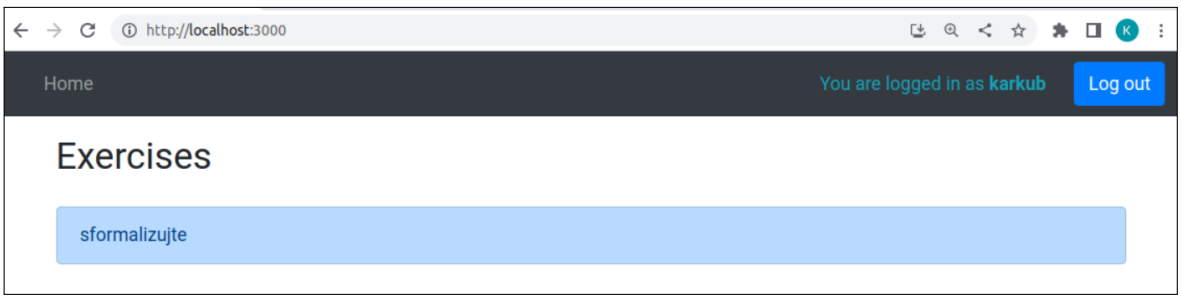



Obr. 2.3: Pôvodný komponentový diagram

## 2.4 Ukážka aplikácie

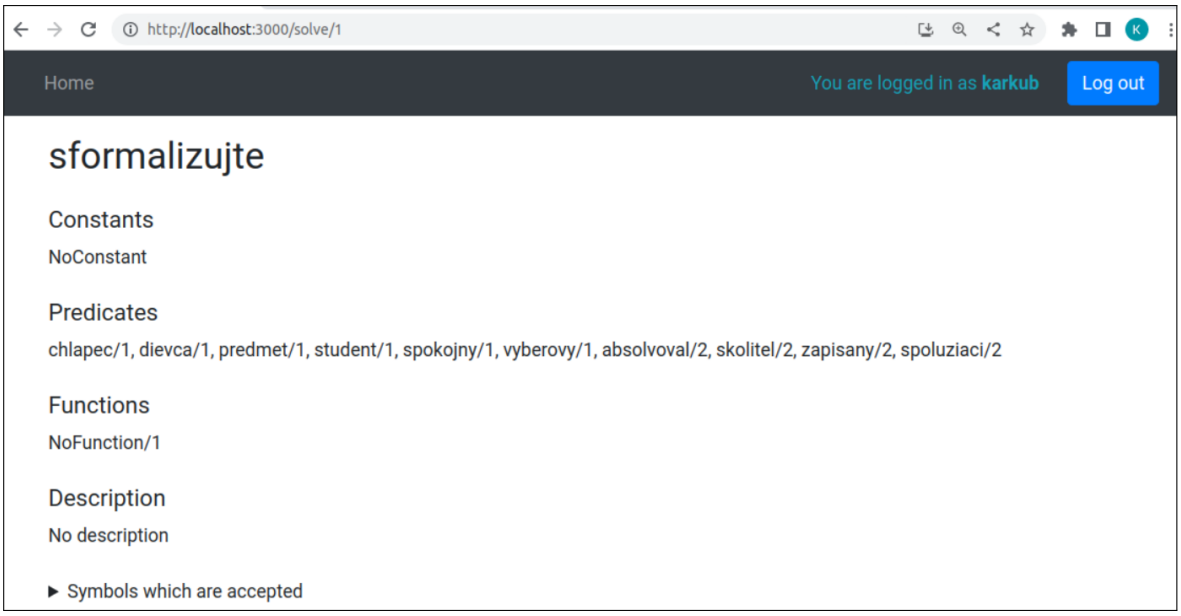
V tejto sekcii si ukážeme rôzne časti používateľského rozhrania aplikácie, po implementácii oboch vyššie spomenutých prác.

Najskôr si ukážeme ako vyzerá proces riešenia cvičenia z pohľadu študenta, pričom pohľad učiteľa sa v tejto časti líši iba obsiahlejším menu. Po prihlásení sa používateľovi zobrazí hlavná stránka, kde sa nachádza zoznam cvičení, ktoré môže riešiť. Ukážku tejto stránky vidíme na obrázku 2.4.



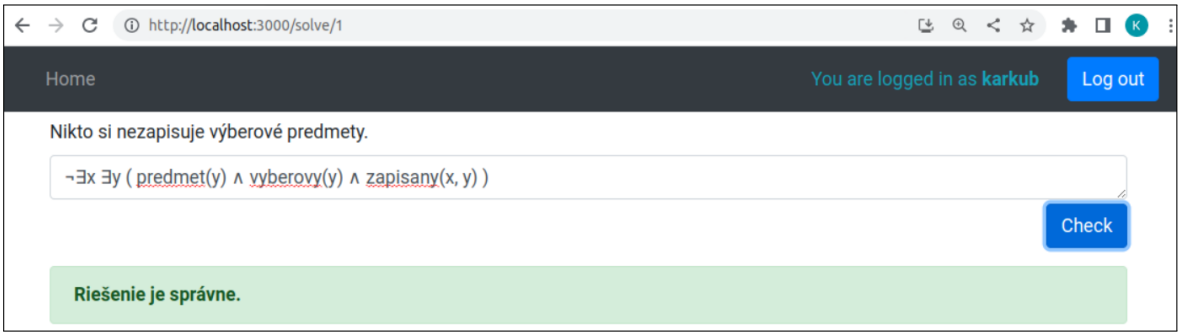

Obr. 2.4: Zoznam cvičení

Po vybratí niektorého cvičenia sa zobrazí jazyk cvičenia, tak ako ukazuje obrázok 2.5. Navyše sa zobrazí zoznam všetkých tvrdení na sformalizovanie.




Obr. 2.5: Cvičenie

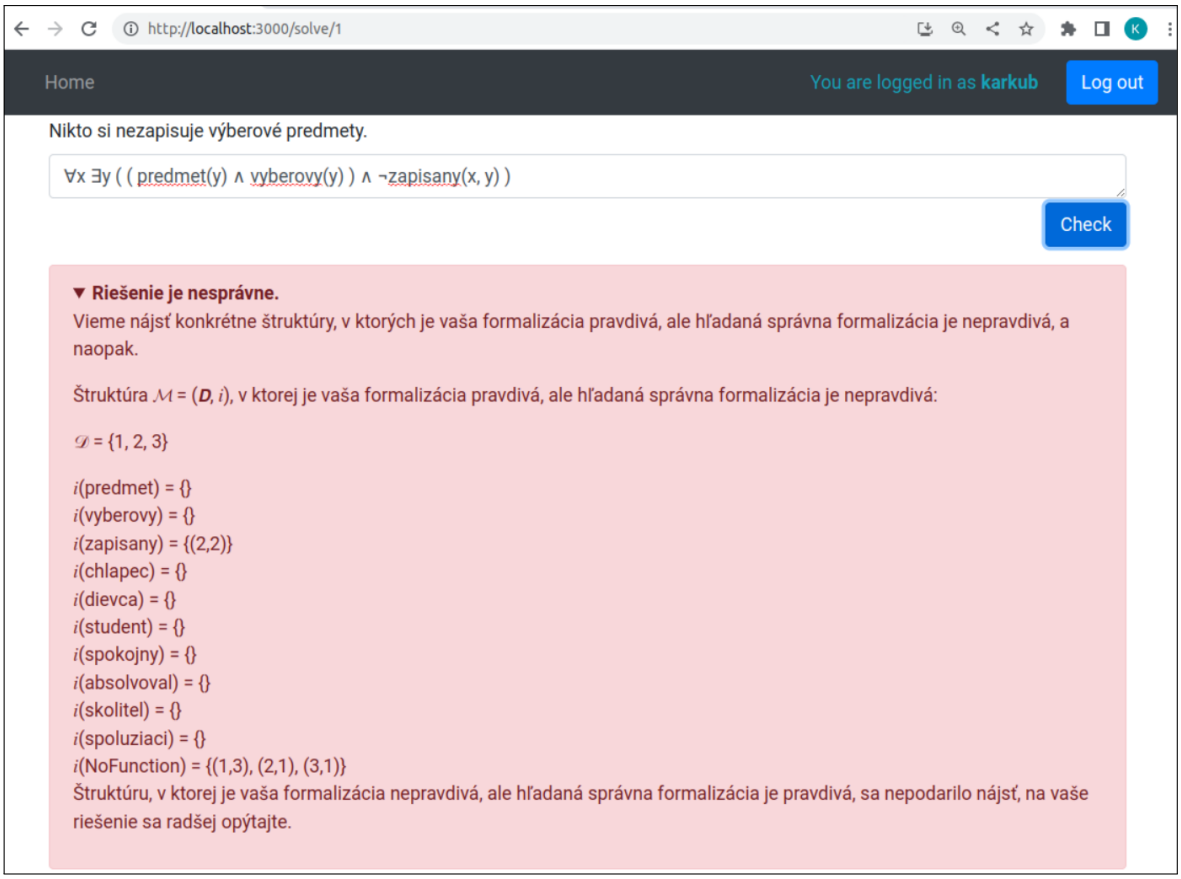
Používateľ následne k nejakému tvrdeniu zadá svoju formalizáciu. Ak je riešenie zapísané korektne, po odoslaní sa overí jeho správnosť. V prípade správnej formalizácie sa zobrazí správa, že riešenie bolo správne, ako vidíme na obrázku 2.6.



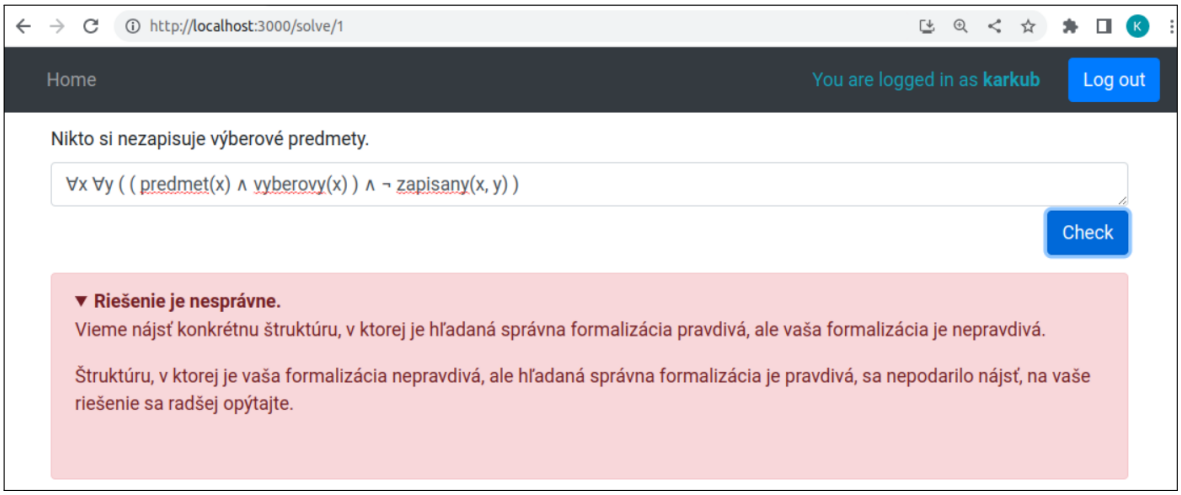

Obr. 2.6: Správne riešenie

Ak riešenie nie je správne, taktiež sa o tom zobrazí správa. Navyše, ak sa dokazovaču Vampire podarilo nájsť kontrapríklady, sa zobrazia aj štruktúry, ako ukazuje obrázok 2.7. V opačnom prípade sa študentovi odporučí spýtať sa na detaily vyučujúceho. Ako vidíme na obrázkoch 2.8 a 2.9, v týchto prípadoch sa dokazovaču nepodarilo nájsť žiadne vhodné štruktúry. Študent teda nemá žiadnu formu spätnej väzby, ktorá by mu napomohla v riešení úlohy.

Učiteľ môže okrem riešenia úloh aj pridávať a editovať cvičenia, spravovať administrátorské práva, a taktiež si môže zobraziť postup študentov v riešení jednotlivých úloh. Nás predovšetkým zaujíma posledné z vymenovaných. Najskôr sa v príslušnej sekcií zobrazí zoznam všetkých cvičení, spolu s počtom študentov, ktorí sa pokúsili riešiť dané cvičenie, ako môžeme vidieť na obrázku 2.10. Obrázok 2.11 ukazuje, že po

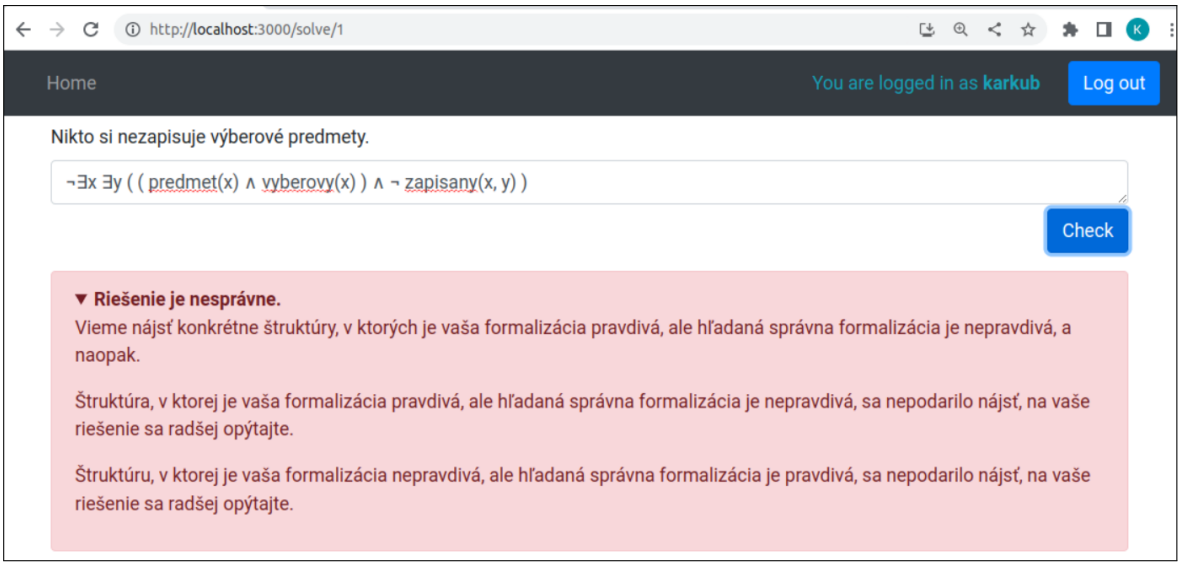



Obr. 2.7: Nesprávne riešenie s výpisom kontrapríkladu – štruktúry




Obr. 2.8: Nesprávne riešenie bez kontrapríkladu - štruktúry

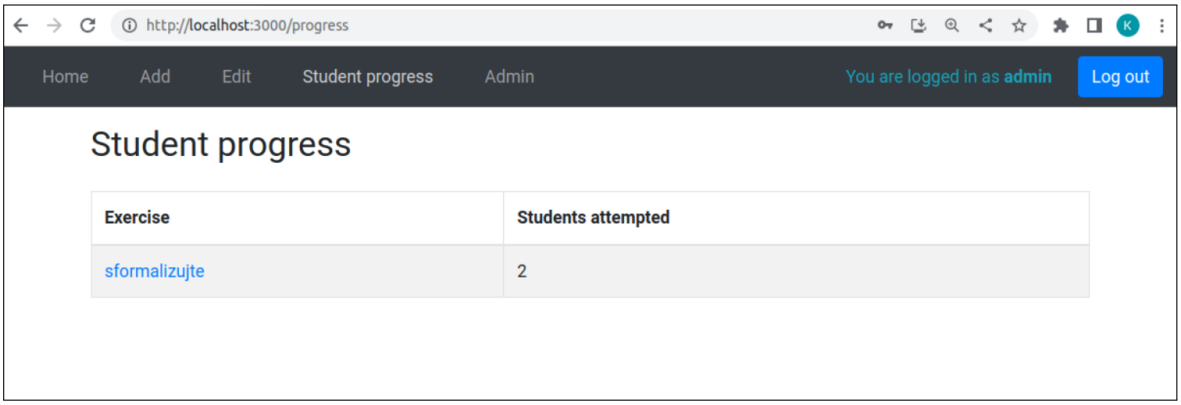
zvolení niektorého cvičenia sa zobrazí zoznam všetkých používateľov, ktorí riešili dané cvičenie. Navyše sa zobrazí aj menšia štatistika ich riešení, presnejšie počet tvrdení, ktoré v danej úlohe riešili a koľko z nich aj úspešne vyriešili, celkový počet pokusov a počet úspešných pokusov. Taktiež sa zobrazuje aj dátum posledného pokusu, spolu s informáciou, či bol úspešný. Po vybratí niektorého používateľa sa zobrazia všetky



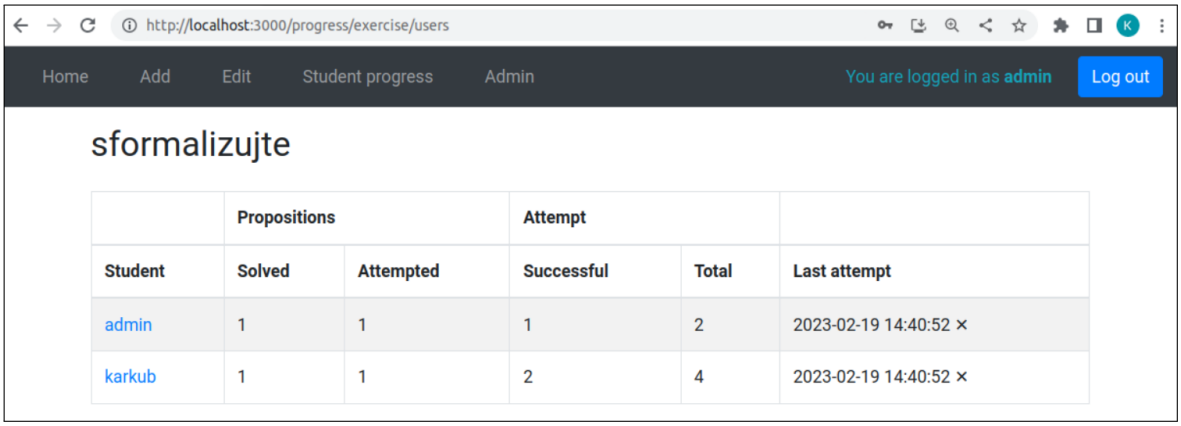

Obr. 2.9: Nesprávne riešenie bez kontraprikladu - štruktúry

jeho riešenia k daným tvrdeniam cvičenia, spolu s dátumom a informáciou, či bolo riešenie správne, ako ukazuje obrázok 2.12.

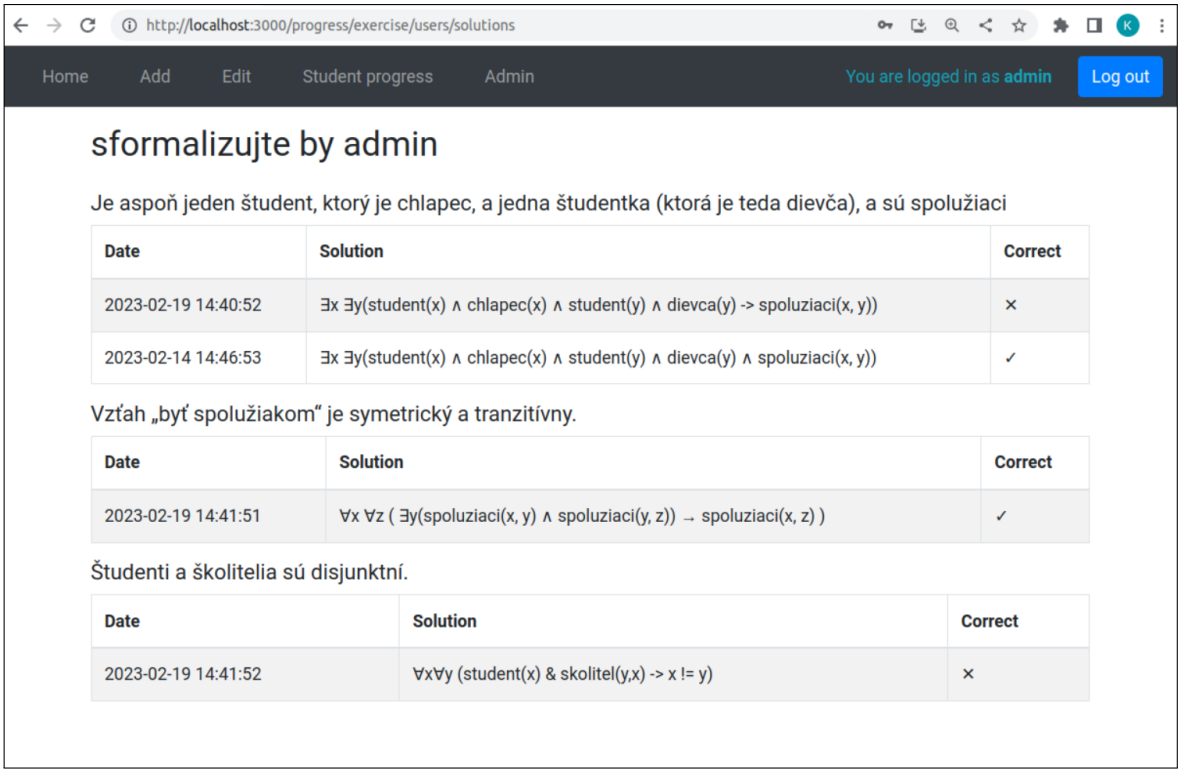
Môžeme si však všimnúť, z obrázkov 2.11 a 2.12, že výpočet štatistiky nie je správny. V skutočnosti používateľ *admin* riešil tri tvrdenia, pričom správne vyriešil dve z nich. Taktiež je chybný aj počet celkových a úspešných pokusov. Bude teda potrebné opraviť získavanie týchto informácií.




Obr. 2.10: Štatistika riešení cvičení




Obr. 2.11: Zoznam riešiteľov daného cvičenia




Obr. 2.12: Postup študenta v riešení cvičenia

# Kapitola 3

# Použité technológie

V tejto kapitole si predstavíme technológie, ktoré boli využité pri predošlej tvorbe aplikácie, a ktoré využijeme aj pri našej implementácií.

Aplikáciu rozdeľujeme, podľa architektúry klient-server, na stranu klienta, ktorá odosiela požiadavky, a na stranu servera, ktorá požiadavky spracúva. Na tvorbu rozhrania klienta bola použitá knižnica React a stav aplikácie manažuje knižnica Redux. Na strane servera bol použitý framework Express. Na kontrolu správnosti formalizácií, ako aj na hľadanie štruktúr ako kontrapríkladov, bol použitý automatický dokazovač Vampire.

Aplikácia využíva, okrem vyššie spomenutých technológií, aj štandardný databázový systém PostgreSQL, na ukladanie cvičení, riešení, či používateľov.

## 3.1 Express

Express [5] je JavaScriptový framework slúžiaci na tvorbu serverových častí aplikácií. Môžeme ním definovať rôzne cesty spolu s funkciami, ktoré spracovávajú HTTP požiadavky na danej ceste. Pomocou metód `GET`, `POST`, `PUT` a `DELETE` môžeme navyše špecifikovať, ako sa majú dané požiadavky spracovať. Express taktiež umožňuje využitie rôznych middleware funkcií, ktoré sa vykonávajú medzi prijatím a spracovaním požiadavky. Tieto funkcie môžu slúžiť, napríklad na logovanie, autentifikáciu, či autorizáciu.

Pomocou Express triedy Router, môžeme zadefinovať cesty a middleware funkcie ku konkrétnym častiam, či modulom aplikácie. V nasledujúcich ukážkach si uvedieme príklady, akým spôsobom sa, v terajšej implementácii, spracovávajú rôzne požiadavky.

Najskôr inicializujeme Express server, ako vidíme na ukážke kódu 3.1. Touto inicializáciou sme určili, že cesty s prefixom `/api/exercises` obsluhuje router definovaný v `./routes/api/exercises`. Takisto sme umožnili využitie autorizačného middleware JSON Web Token – `jwt`. Samotný server sa spustí na danom porte volaním metódy

listen.

```
const express = require('express');
const server = express();
server.use('/api/exercises', require('./routes/api/exercises'));
server.use(jwt({ secret: TOKEN_SECRET, algorithms: ['HS256']}).
   unless({path: ['/logIn', '/logIn/github/auth']}));
server.listen(PORT, () => {
  console.log(`Server started listening on port ${PORT}`);
});
```

Listing 3.1: Inicializácia Express servera

Na získanie zoznamu cvičení, klient odošle HTTP požiadavku na server na cestu `/api/exercises`, použitím metódy `GET`, tak ako vidíme na ukážke kódu 3.2.

```
export const fetchAllExercises = createAsyncThunk(
  'exercises/fetchAllExercises',
  async (_, { rejectWithValue }) => {
    try {
      let response = await fetchData('/api/exercises', 'GET');
      return response;
    } catch (err) {
      return rejectWithValue(err.message);
    }
  }
)
```

Listing 3.2: Odoslanie požiadavky na zísaknie cvičení na server

Ako sme definovali pri inicializácii servera, router v `./routes/api/exercises` na strane servera odchytí `GET` požiadavku na ceste `/api/exercises`, ako môžeme vidieť na ukážke kódu 3.3. Ďalej zavolá databázovú funkciu, na získanie zoznamu cvičení. V prípade úspešného získania zoznamu cvičení, router odošle tieto cvičenia späť na klienta, v opačnom prípade mu pošle kód chyby.

```
router.get('/', async (req, res) => {
    try {
        const previews = await getExercisePreviews();
        if (previews === null) {
            res.sendStatus(500);
            return;
        }
        res.status(200).json(previews);
    } catch (err) {
```

```
        res.sendStatus(500);
    }
});
```

Listing 3.3: Spracovanie požiadavky na zísaknie cvičení na serveri

Pri vytváraní nového cvičenia sa postupuje podobne, ako môžeme vidieť na ukážke kódu 3.4. V prípade, že cvičenie je zapísané korektne, klient odošle HTTP požiadavku na server na cestu `/api/exercises`, tentokrát použitím metódy `POST`.

```
export const addNewExercise = createAsyncThunk(
  'addExercise/addNewExercise',
  async (_, { getState, rejectWithValue }) => {
    let exercise = selectExercise(getState());
    if (!exercise) {
      return rejectWithValue("Exercise contains errors.");
    }
    try {
      let response = await fetchData(
        '/api/exercises', 'POST', exercise
      );
      return response;
    } catch (err) {
      return rejectWithValue(err.message);
    }
  }
);
```

Listing 3.4: Odoslanie požiadavky na pridanie nového cvičenia na server

Router na strane servera odchytí `POST` požiadavku na ceste `/api/exercises`, ako môžeme vidieť na ukážke kódu 3.5. Ak je cvičenie zapísané korektne, zavolá sa databázová funkcia, ktorá riešenie uloží. V prípade úspešného uloženia, router odošle výstup z databázovej funkcie späť na klienta. Inak mu pošle kód chyby.

```
router.post('/', authAdmin, async (req, res) => {
    try {
        let exercise = req.body;
        if (checkExercise(exercise)) {
            await saveExercise(exercise);
        }
        res.status(201).json(exercise);
    } catch (err) {
        res.sendStatus(500);
```

```
    }
});
```

Listing 3.5: Spracovanie požiadavky na zísaknie cvičení na serveri

Všimnime si, že aj napriek tomu, že klient odošle rôzne požiadavky na rovnakú cestu, router na strane servera spracuje každú požiadavku rôzne. Dôvodom je využitie iných metód, v prvom prípade sa využila `GET` metóda a v druhom `POST`.

## 3.2 React

React [6] je JavaScriptová knižnica, ktorá slúži na tvorbu používateľských rozhraní webových aplikácií. Hlavnou výhodou Reactu je jeho efektivita. Pri zmene dát React, pomocou stavov, zistí, ktoré časti rozhrania je potrebné aktualizovať a iba tie aktualizuje. Týmto sa predchádza zbytočnému spomaľovaniu výkonu aplikácie. Jednotlivé časti používateľského rozhrania vytvárame deklaratívnym spôsobom a nazývame ich komponenty.

*Komponent* je znovupoužiteľný enkapsulovaný kód reprezentujúci časť používateľského rozhrania. Každý komponent môže obsahovať rôzne elementy jazyka HTML, prípadne ďalšie komponenty. Komponenty môžu taktiež spravovať svoj stav, pomocou štruktúry `state`, ktorá definuje, akým spôsobom sa daný komponent vykresľuje. Každý komponent môže taktiež dostať nejaké dáta z rodičovského komponentu pomocou argumentu `props`. React navyše umožňuje funkcionálnym komponentom pristupovať k rôznym informáciám prostredníctvom hákov. Hák, resp. *hook*, je funkcia, ktorá sa volá z funkcionálneho komponentu a umožňuje mu spravovať jeho vnútorný stav, ale aj pristupovať k rôznym ďalším funkciám Reactu. Medzi najpoužívanejšie React Hooky patrí, napríklad `useState`, ktorý umožňuje spravovať stav vrámci nejakého funkcionálneho komponentu, alebo `useEffect`, pomocou ktorého môžeme, napríklad získať dáta z API po renderovaní komponentu. Použitie `useState` háku môžeme vidieť na ukážke kódu 3.6.

```
function Example() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times.</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
```

```
}
```

Listing 3.6: Použitie React Hooks

Na routovanie v aplikáciách vytvorených za pomoci frameworku React sa využíva knižnica React Router [7]. Táto knižnica poskytuje manažovanie navigácie medzi jednotlivými komponentami aplikácie. Hlavným komponentom je `BrowserRouter`, ktorý sa využíva v najvyššej úrovni hierarchie komponentov, vďaka čomu je zaistená synchronizácia URL adries a vykreslených komponentov. Ďalším dôležitým komponentom je `Route`, ktorý priraďuje komponent, ktorý sa má vykresliť, k určitej URL adrese. V predošlej implementácii boli navyše vytvorené vlastné Route komponenty. Prvý z nich je `ProtectedRoute`, ktorý vykreslí príslušný komponent iba ak je používateľ prihlásený a druhý `AdminRoute` vykreslí komponent iba administrátorovi.

Na nasledujúcej ukážke kódu 3.7 vidíme použitie rôznych Route komponentov v doterajšej implementácii, pričom v atribúte `component` je priradený konkrétny komponent, ktorý sa má zobraziť na danej ceste, zapísanej v atribúte `path`. Atribút `exact` určuje, že daný komponent sa má vypísať iba na danej ceste a nie aj na rôznych podstránkach danej cesty.

```
<BrowserRouter basename={BASE_NAME}>
    ...
    <Switch>
        <ProtectedRoute exact path="/" component={ExerciseList}/>
        <AdminRoute exact path="/admins" component={UserList}/>
        <Route exact path="/login" component={LoginForm}/>
        ...
    </Switch>
    ...
</BrowserRouter>
```

Listing 3.7: Príklad použitia React Router

## 3.3 Redux a RTK Query

Redux [8] je taktiež JavaScriptová knižnica, ktorá slúži na manažovanie stavu aplikácie. Stav celej aplikácie sa uchováva v jednom úložisku, nazývanom `store`, pričom inicializáciu úložiska našej aplikácie môžeme vidieť na ukážke kódu 3.8. Vďaka tomu môžu jednotlivé komponenty pristupovať k rôznym stavom bez toho, aby bolo potrebné navzájom prepojiť jednotlivé komponenty. Takisto sa tým uľahčuje proces aktualizovania údajov, keďže stačí aktualizovať tieto dáta iba na jednom mieste a komponenty. Stav aplikácie môžeme jednoducho meniť deklaratívnym spôsobom, kedy opíšeme čo

sa má stať s rôznymi údajmi pri rôznych akciách, ale samotné aktualizovanie stavu za náš vyrieši Redux. *Akcie* teda popisujú rôzne udalosti, ktoré môžu mať za následok zmenu stavu. Funkcie, ktoré na základe aktuálneho stavu aplikácie a rôznych akcií generujú nový stav, sa nazývajú *reducery*. Redux takisto umožňuje použitie rôznych *middleware* funkcií, ktoré sa vykonávajú medzi odoslaním nejakej akcie a jej následným spracovaním reducermi. V doterajšej implementácii sa middleware využíva, napríklad pri prihlasovaní používateľa pomocou portálu GitHub.

```
export default configureStore({
  reducer: {
    addExercise: addExerciseReducer,
    exercises: exercisesReducer,
    propositions: progressPropositionsReducer,
    allUsers: adminsReducer,
    solveExercise: solveExerciseReducer,
    user: userReducer
  }
})
```

Listing 3.8: Inicializácia úložiska v našej aplikácii

RTK Query [9] je nástroj využívaný na správu dátových požiadaviek a stavu aplikácie za pomoci Reduxu. Pri použití RTK Query nemusíme definovať rôzne akcie, middleware, či reducery, ale v tzv. *API slice* definujeme, pomocou `createApi`, nejaké endpointy a na vykonávanie požiadaviek využijeme hooky.

*Endpoint* je definícia požiadavky. Každý endpoint obsahuje URL adresu koncového bodu, ktorý vykonáva danú požiadavku, HTTP metódu, ktorá určuje typ požiadavky, a rôzne ďalšie parametre, ako napríklad meno endpointu, pomocou ktorého ho môžeme identifikovať. Vo voliteľnom parametri *body* môžeme definovať dáta, ktoré sa majú odoslať spolu s požiadavkou. Dáta, ktoré získame z rôznych požiadaviek môžeme označiť rôznymi značkami pomocou `providesTags`. Ak niektoré požiadavky neskôr zmenili tieto dáta, môžeme ich v danom endpointe označiť pomocou `invalidatesTags` ako neaktuálne, čím zaručíme aktualizovanie príslušných dát ich znovunačítaním zo servera. Na základe definovaných endpointov, RTK Query automaticky generuje háky, ktoré fungujú podobne ako sme opísali v sekcii 3.2. Pomocou týchto hákov môžeme jednoducho pristupovať k dátam zo servera.

Pri využití RTK Query je inicializácia úložiska mierne odlišná, oproti využitiu bežného Reduxu. Táto inicializácia s využitím RTK Query je znázornená na nasledujúcej ukážke kódu 3.9.

```
export const store = configureStore({
  reducer: {
```

```
    [myApi.reducerPath]: myApi.reducer,
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(myApi.middleware),
})
```

Listing 3.9: Inicializácia úložiska s využitím RTK Query

V ukážke kódu 3.10 vidíme ako sa vyrába RTK Query slice. Môžeme si všimnúť zadefinovanie značiek v `tagTypes` a následne prácu s nimi v jednotlivých endpointoch pomocou `providesTags` a `invalidatesTags`. V porovnaní s bežnými Redux slices, ktoré sú znázornené na ukážkach kódov 3.2 a 3.4, vidíme, že využitie RTK Query je oveľa stručnejšie a bez výskytu duplicitného kódu.

```
export const myApi = createApi({
  reducerPath: 'myApi',
  baseQuery: fetchBaseQuery({ baseUrl: '/exercises' }),
  tagTypes: ['Exercises'],
  endpoints: (builder) => ({
    getExercises: build.query({
      query: () => ({
        url: `/`,
        method: 'GET'
      }),
      providesTags: ['Exercises'],
    }),
    addExercise: build.mutation({
      query: (body) => ({
        url: `/`,
        method: 'POST',
        body,
      }),
      invalidatesTags: ['Exercises'],
    }),
  }),
})
export const {
    useGetExercisesQuery,
    useAddExerciseMutiation
} = myApi
```

Listing 3.10: Tvorba RTK Query slice

Na ukáže kódu 3.11 vidíme využitie háku `useGetExercisesQuery`. K dátam získaným zo samotnej požiadavky vieme pristupovať pomocou konštanty `data`. RTK Query háky navyše ponúkajú aj ďalšie užitočné informácie, napríklad či prebehlo získanie údajov úspešne, prípadne, či nastala nejaká chyba, ale aj mnohé ďalšie.

```
export default function App() {
  const { data, isSuccess, isError, error } =
     useGetExercisesQuery()
  ...
}
```

Listing 3.11: Použitie RTK Query háku

## 3.4 Vampire

Vampire [10] je automatický dokazovač logických formúl. V našej aplikácii sa Vampire využíva na overovanie ekvivalencie dvoch formúl. Takisto sa využíva aj na získanie kontrapríkladu – štruktúry, v prípade, že formuly nie sú navzájom ekvivalentné. Vampire dostáva na vstup dve formuly, ktorých vzájomnú ekvivalenciu chceme overiť, v TPTP formáte, pričom prevod na tento formát v pôvodnej aplikácii zabezpečuje funkcia `parseFormalization` v `helpers/checks.js` na strane servera. Formuly v TPTP formáte môžeme potom uložiť do súboru s príponou `.p`, napríklad `subor.p`. Samotné volanie dokazovača potom prebieha pomocou príkazu `vampire subor.p`. V prípade, že výstup potom obsahuje reťazec `Refutation`, dané formuly zo vstupu sú ekvivalentné. Ak naopak obsahuje reťazec `Satisfiable`, Vampire našiel kontrapríklad, ktorý potom môžeme vypísať pomocou príkazu `vampire subor.p -sa fmb`.

Na prácu s Vampire sa využívajú viaceré abstraktné funkcie na serveri, ktoré majú rôzne využitie. Tieto funkcie boli vytvorené v predošlých implementáciách aplikácie. Na kontrolu ekvivalencie dvoch formúl, napríklad formúl `A` a `B`, slúži funkcia `vampire`, ktorá sa nachádza v súbore `helpers/vampire.js`. Túto funkciu je potrebné zavolať dvakrát. Najskôr teda zavoláme `vampire(A, B, TIME_LIMIT)` na zistenie, či z formuly `A` vyplýva formula `B`, a potom vymeníme poradie formúl a znova zavoláme `vampire(B, A, TIME_LIMIT)`. Formuly `A` a `B` sú potom ekvivalentné iba v prípade, že obe volania funkcie vrátia reťazec `OK`.

# Kapitola 4

# Požiadavky a návrh rozšírenia aplikácie

V tejto kapitole zadefinujeme požiadavky na rozšírenie predošlej aplikácie a navrhneme ich postup implementácie.

## 4.1 Požiadavky na aplikáciu

Medzi základné požiadavky na naše rozšírenie už existujúcej aplikácie patrí:

- zoskupovať chybné riešenia na základe ekvivalencie,
- možnosť pridávať nápovede pre učiteľa,
- zobrazovať nápovede k danej chybe,
- možnosť ohodnotiť užitočnosť nápovedí,
- vyhodnotiť užitočnosť nápovedí vo výučbe,
- zrefaktorovať predošlý kód.

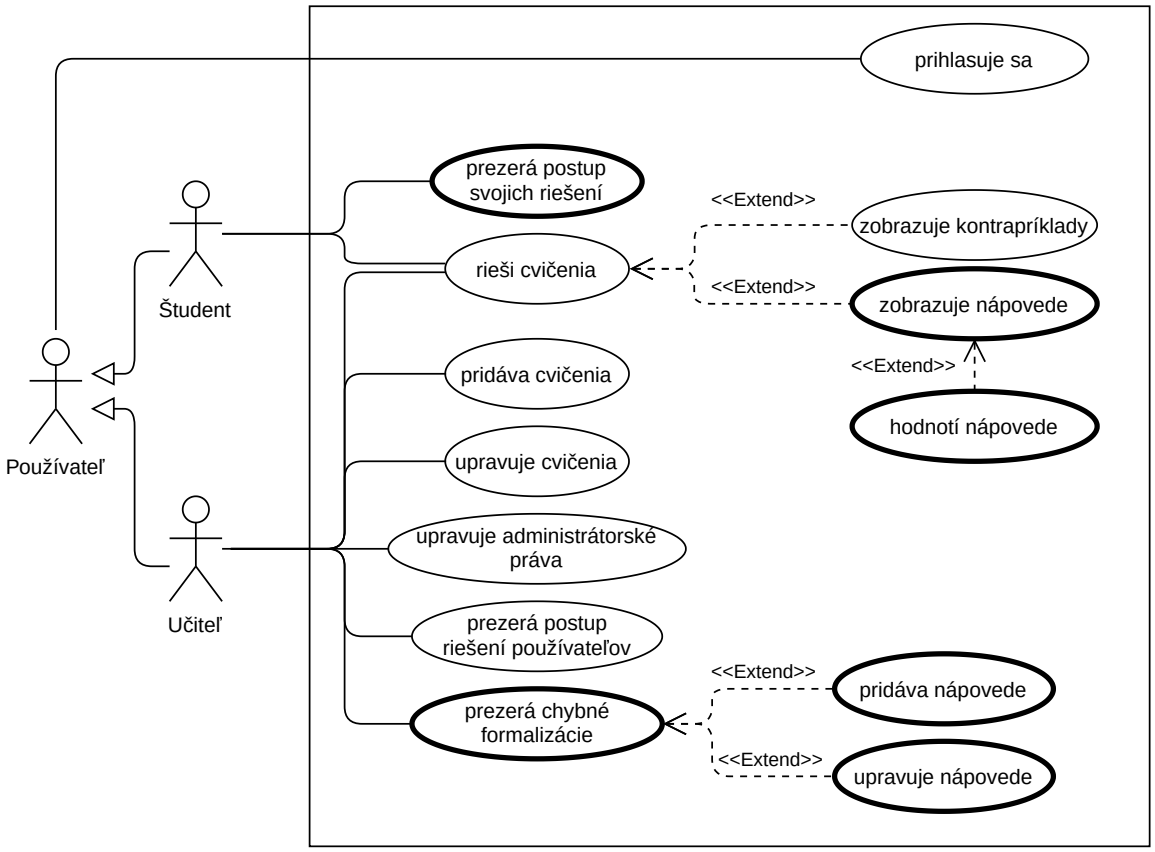
Hlavnou požiadavkou je implementácia nového spôsobu spätnej väzby pomocou nápovedí. Učitelia budú môcť pridávať nápovede k jednotlivým chybným formalizáciám, ktoré sa následne budú zobrazovať študentom, keď odošlú tieto chybné formalizácie na kontrolu.

Na tieto účely je najskôr potrebné zoskupovať chybné riešenia, ktoré sú si ekvivalentné. Učiteľ následne bude môcť pridať nápoveď niektorej triede ekvivalentných chybných riešení, a táto nápoveď sa potom zobrazí pri každom chybnom riešení, ktoré je ekvivalentné danej triede.

Ďalšou požiadavkou je implementovať možnosť ohodnotiť danú nápoveď. Učitelia tak budú mať prehľad, ktoré nápovede vnímajú študenti užitočnejšie ako iné a na základe toho potom prispôsobiť tieto nápovede.

## 4.2 Návrh aplikácie

Na základe vyššie opísaných požiadaviek môžeme navrhnúť nový use case diagram, ktorý bude obsahovať všetky aktivity z pôvodného diagramu (obrázok 2.1), ako aj nové funkcie, ktoré sú na obrázku 4.1 vyznačené hrubo. Implementáciu jednotlivých požiadavok si popíšeme a navrhneme nižšie. Taktiež je potrebné spracovať nápady na vylepšenie doterajšej implementácie opísané v sekcii 2.3.




Obr. 4.1: Aktuálny use case diagram

### 4.2.1 Zoskupovanie chybných riešení

Hlavnou požiadavkou je zoskupovanie chybných riešení. Na toto zoskupovanie budeme používať vzťah ekvivalencie. Keďže na overovanie ekvivalencie medzi dvoma formulami sa už využíva dokazovač Vampire, využijeme ho aj na toto overovanie.

Akonáhle študent odošle svoju formalizáciu na kontrolu, dokazovač overí, či je jeho formalizácia ekvivalentná správnemu riešeniu, a naopak, či je správne riešenie ekvivalentné tomu študentovmu. Ak niektorá z týchto vlastností neplatí, riešenie nie je správne. V takom prípade bude potrebné získať všetkých reprezentantov chybných formalizácií daného tvrdenia a porovnať, či je študentovo riešenie ekvivalentné niektorému z reprezentantov. Ak takého reprezentanta nájdeme, toto riešenie bude patriť do jeho

ekvivalenčnej triedy a do databázy uložíme toto riešenie s cudzím kľúčom odkazujúcim na id reprezentanta chybnej formalizácie. Ak ešte v databáze nemáme žiadnych reprezentantov, prípadne neexistuje žiadny ekvivalentný študentovej formalizácii, pridáme toto riešenie do príslušnej tabuľky a riešenie uložíme s cudzím kľúčom, odkazujúcim na svoje id v príslušnej tabuľke.

### 4.2.2 Pridávanie nápovedí

Učiteľ si bude môcť zobraziť všetkých reprezentantov chybných formalizácii k jednotlivým tvrdeniam, pričom pri každom reprezentantovi sa budú zobrazovať aj všetky chybné formalizácie nemu ekvivalentné. Učiteľ bude mať možnosť priradiť nejakej skupine zlých formalizácií nejakú nápoveď, ktorá bude slúžiť ako pomoc pre študentov. Pri každej nápovedi sa bude zobrazovať počet jej zobrazení a jej hodnotenie. Na základe takejto štatistiky budú môcť učitelia upraviť, či sa bude daná nápoveď zobrazovať študentom, alebo bude skrytá.

### 4.2.3 Zobrazovanie a hodnotenie nápovedí

Študentom sa bude zobrazovať nápoveď, ktorú učitelia pridali k jednotlivým reprezentantom chybných formalizácií, akonáhle bude ich chybné riešenie ekvivalentné niektorému reprezentantovi chybnej formalizácie. Používateľ teda odošle svoje riešenie na kontrolu, tak ako doteraz, pričom okrem kontrapríkladu, môže dostať aj nejakú nápoveď od učiteľov, ktorá bude vysvetľovať chyby v danom riešení. Tieto nápovede si bude môcť postupne zobrazovať a každú samostatne ohodnotiť, prostredníctvom palca hore, či dole, podľa toho, či sa mu zdá daná nápoveď užitočná, alebo nie.

### 4.2.4 Zobrazovanie postupu vlastných riešení študentovi

Tak ako je učiteľom umožnené zobrazovať riešenia študentov k jednotlivým úlohám, sprístupníme túto funkciu aj študentom. Študent sa bude môcť prekliknúť na príslušnú podstránku cez menu, kde sa mu zobrazí zoznam úloh, ktoré riešil. Pri každej úlohe sa bude zobrazovať aj štatistika, presnejšie počet tvrdení, ktoré študent riešil, počet tvrdení, ktoré úspešne vyriešil, počet celkových a počet úspešných pokusov. Po vybratí niektorej úlohy sa mu zobrazia vŠetky jeho riešenia jednotlivých tvrdení, ktoré patria k vybranej úlohe.

### 4.2.5 Refaktorovanie predošlého kódu

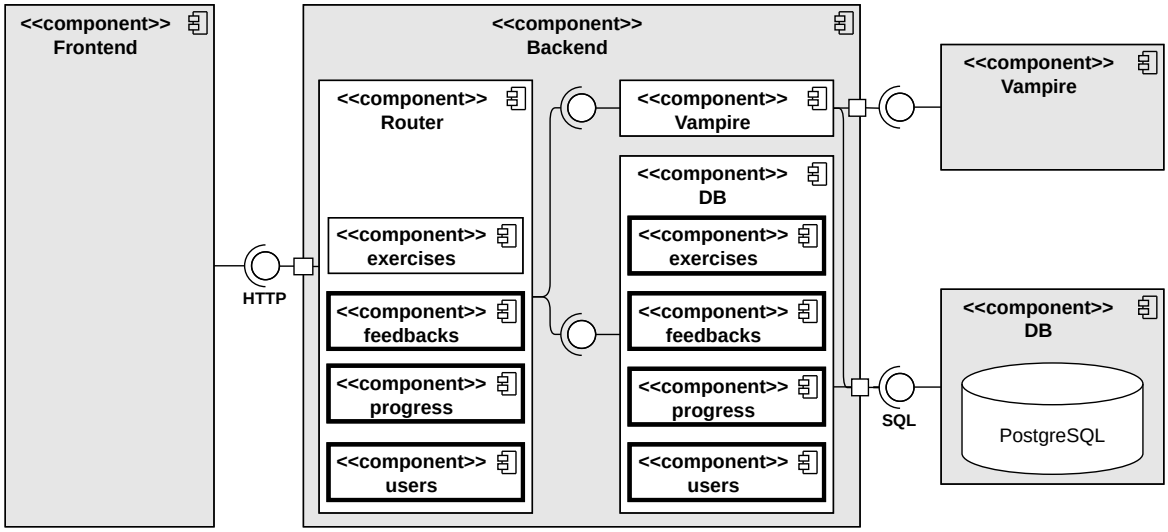
Ako sme si ukázali v sekcii 2.3, doterajšia implementácia nie je ideálna, a preto je potrebné upraviť viaceré časti kódu.

Aktuálne sú na strane servera dva moduly `db/getData.js` a `db/saveData.js`, v ktorých sú rozdelené funkcie obsluhujúce databázové požiadavky, podľa toho, či z databázy čítame, alebo do nej zapisujeme. Bude však lepšie rozdeliť tieto funkcie do modulov podľa typu dát, s ktorými narábajú. Funkcie, ktoré sa využívajú pri práci s cvičeniami, či už na zobrazovanie, pridávanie, editovanie, alebo riešenie cvičení, presunieme do samostatného modulu. Rovnako presunieme do samostatných modulov aj funkcie, ktoré sa týkajú prihlasovania používateľov, a funkcie, ktoré slúžia na zobrazovanie postupu študentov v riešení. Aby sme predišli duplicite kódu využijeme pomocné funkcie, ktoré nebudeme exportovať, ale využijeme ich iba interne, vrámci daného modulu.

Takisto bude potrebné upraviť cesty, cez ktoré prebieha komunikácia medzi klientom a serverom. Momentálne je na strane servera jeden modul `api/exercises.js`, slúžiaci ako router, ktorý však obsahuje aj cesty, ktoré sa priamo netýkajú len cvičení. Napríklad, na účely prihlasovania sa využívajú cesty `/api/exercises/logIn/github/auth` a `/api/exercises /authentication/logIn/admin`. Takáto štruktúra ciest však nie je ideálna, a preto prácu s rôznymi typmi údajov rozdelíme do samostatných modulov s príslušnými cestami.

Navyše bude potrebné opraviť niektoré časti kódu. Ako sme si mohli všimnúť na ukážkach aplikácie v sekcii 2.4, výpočet štatistiky v zobrazovaní postupu študentov nie je správny a je potrebné ho opraviť.

Nový komponentový diagram je navrhnutý na obrázku 4.2, pričom nové a upravené komponenty z pôvodného diagramu (obrázok 2.3) sú zvýraznené hrubo.




Obr. 4.2: Aktuálny komponentový diagram

### 4.2.6 Úprava databázového modelu

Taktiež je potrebné upraviť existujúcu databázu a pridať nové tabuľky, aby sme mohli implementovať všetky požiadavky.

Vytvoríme novú tabuľku `bad_formalizations` na ukladanie reprezentantov chybných formalizácií, ktorá bude obsahovať stĺpce:

- `bad_formalization_id` - (primárny kľúč) id reprezentanta chybnej formalizácie
- `bad_formalization` - text chybnej formalizácie
- `proposition_id` - (cudzí kľúč) id tvrdenia na sformalizovanie.

Následne každé riešenie budeme ukladať do tabuľky `solutions` s dvoma novými cudzími kľúčmi:

- `formalization_id` - id správnej formalizácie
- `bad_formalization_id` - id reprezentanta chybnej formalizácie, v prípade správneho riešenia bude mať hodnotu `null`.

Na ukladanie jednotlivých nápovedí k chybným formalizáciám budeme využívať tabuľku `feedbacks`, ktorá bude obsahovať stĺpce:

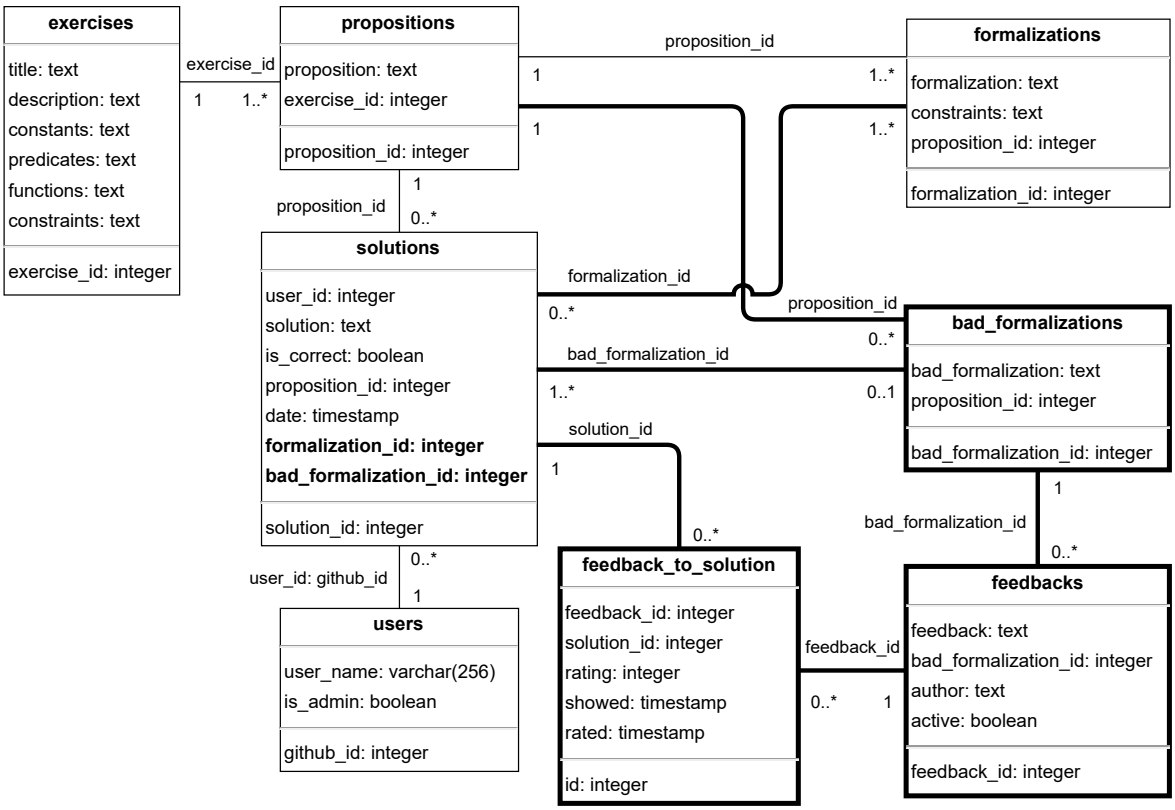
- `feedback_id` - (primárny kľúč) id nápovede
- `feedback` - text nápovede
- `bad_formalization_id` - (cudzí kľúč) id reprezentanta chybnej formalizácie, ku ktorému patrí daná nápoveď
- `author` - používateľské meno autora nápovede
- `active` - informácia o tom, či sa nápoveď zobrazuje študentom, alebo je skrytá.

Keďže jedna nápoveď sa môže zobraziť pri viacerých chybných riešeniach, a taktiež pri jednom chybnom riešení sa môže zobraziť viacero nápovedí, medzi tabuľkami `solutions` a `feedbacks` je vzťah *many-to-many*. Preto vytvoríme ešte jednu tabuľku `feedback_to_solution`, ktorá bude spájať konkrétne riešenie a konkrétnu nápoveď. V tejto tabuľke budeme zároveň ukladať hodnotenie danej spätnej väzby a ďalšie užitočné informácie. Tabuľka bude teda obsahovať stĺpce:

- `id` - (primárny kľúč) id
- `feedback_id` - (cudzí kľúč) id nápovede
- `solution_id` - (cudzí kľúč) id riešenia

- `rating` - hodnotenie nápovede, v prípade, že ju študent ohodnotil ako užitočnú, má hodnotu 1, ak ako neužitočnú -1 a ak ju neohodnotil 0
- `showed` - dátum kedy sa používateľovi zobrazila daná nápoveď
- `rated` - dátum kedy používateľ ohodnotil danú nápoveď.

Všetky vyššie popísané zmeny v doterajšej schéme databázy (obrázok 2.2) sú vyznačené hrubo na obrázku 4.3.




Obr. 4.3: Aktuálna schéma databázy

Aby sme mohli upraviť databázu, bude takisto potrebné premigrovať údaje, ktoré sú v nej už uložené. Najskôr upravíme a vytvoríme nové tabuľky v databáze, pričom doplnené atribúty budú mať zatiaľ hodnotu `null`. Ďalej postupne prejdeme všetky riešenia a budeme porovnávať, či je chybné riešenie ekvivalentné niektorému reprezentantovi z tabuľky `bad_formalizations`. Ak nie, toto riešenie pridáme do spomenutej tabuľky a príslušne upravíme atribút `bad_formalization_id` v tabuľke `solutions`. Ak také ekvivalentné riešenie nájdeme, do tabuľky `bad_formalizations` už nemusíme nič pridávať a iba upravíme atribút `bad_formalization_id` v tabuľke `solutions`, podľa jeho id.

# Kapitola 5

# Implementácia

V tejto kapitole si ukážeme ako sme implementovali požiadavky podľa návrhu.

## 5.1 Refaktorovanie predošlého kódu

Z databázových modulov `db/getData.js` a `db/saveData.js` na strane servera sme rozdelili funkcie do nových modulov podľa typu dát, s ktorými pracovali, ako sme navrhli v sekcii 4.2.5. Namiesto týchto modulov sme teda vytvorili tri nové moduly `db/exercises.js`, `db/progress.js` a `db/users.js`, ktoré postupne obsahujú funkcie narábajúce s cvičeniami, postupom študentov v riešení cvičení a s používateľmi. Takisto sme často opakujúce sa časti kódu vyňali do privátnych funkcii, ktoré ďalej neexportujeme, ale využívame ich iba interne v danom module.

Takisto sme presunuli databázové transakcie dovnútra jednotlivých funkcií, pokiaľ sa teda nejedná o privátnu funkciu. Predtým funkcie v module `api/exercises.js`, ktorý slúži na obsluhu ciest a spracovanie požiadavok zo strany klienta, najskôr začali transakciu a potom volali jednotlivé databázové funkcie. Tento modul sme taktiež rozdelili na viacero modulov, pričom pre prácu s cvičeniami sa využíva prefix cesty `api/exercises` a modul `api/exercises.js`. Na zobrazovanie postupu študentov v riešení cvičení využívame cestu s prefixom `api/progress` a modul `api/progress.js` a pri práci s používateľmi používame prefix `api/users` a modul `api/users.js`. Jednotlivé funkcie v týchto moduloch už neobsahujú databázové transakcie, ale volajú potrebné funkcie z databázových modulov, ktoré sú už transakčne izolované.

## 5.2 Zoskupovanie chybných riešení

Na účely zoskupovania chybných riešení sme najskôr upravili databázu, tak ako sme navrhli v sekcii 4.2.6. Spustením `db/migrations/001-add-bad-formalizations.sql` súboru vytvoríme tabuľku `bad_formalizations`, ktorá slúži na zoskupovanie chybných

formalizácií. Navyše do tabuľky `solutions` sme pridali dva nové stĺpce. Prvý z nich je `formalization_id`, ktorý slúži na prepojenie konkrétneho riešenia so správnou formalizáciou. Druhým je `bad_formalization_id`, ktorý slúži na prepojenie konkrétneho riešenia a reprezentanta chybnej formalizácie.

Keďže v databáze, ktorá je v prevádzke spolu s aplikáciou, už máme uložené nejaké dáta, museli sme ich prispôsobiť novej schéme databázového modelu. Nové atribúty v tabuľke `solutions` sme najskôr naplnili hodnotou `null`, ktorú sme neskôr zmenili. Pomocou migračného skriptu `db/migrations/001-add-bad-formalizations.js` sme postupne prechádzali každé riešenie a doplnili sme k nemu id správnej formalizácie daného tvrdenia. Ďalej, v prípade, že riešenie nebolo správne, sme porovnali, či už existuje nejaký reprezentant chybnej formalizácie k danému tvrdeniu v tabuľke `bad_formalizations` a či mu je aktuálne riešenie ekvivalentné. Ak taký záznam v tabuľke `bad_formalizations` neexistoval, vytvorili sme ho, pričom jeho id sme pridali aj do tabuľky `solutions` ako referenciu na tento nový záznam. V prípade, že sme našli reprezentanta chybnej formalizácie v tabuľke `bad_formalizations`, ktorý je ekvivalentný aktuálnemu riešeniu, jeho id sme pridali ako referenciu k danému riešeniu do tabuľky `solutions`. Takto sme zoskupili všetky doterajšie chybné riešenia z databázy na základe ekvivalencie.

Zoskupovanie nových chybných riešení sme implementovali následovne. Pri odoslaní študentovho riešenia prebieha kontrola, či sa jedná o správne riešenie alebo nesprávne. Využíva sa pri tom dokazovač Vampire, ktorý kontroluje, či je dané riešenie ekvivalentné správnej formalizácii a naopak. Ak zistíme, že riešenie nie je správne, získame všetkých reprezentantov chybných formalizácií, teda všetky záznamy z tabuľky `bad_formalizations` k danému tvrdeniu. Ďalej, taktiež za pomoci dokazovača, budeme kontrolovať, či je niektorý reprezentant ekvivalentný novému riešeniu, a naopak, či je riešenie ekvivalentné danému reprezentantovi. V prípade, ak takéhoto reprezentanta nájdeme, jeho id uložíme ako cudzí kľúč v danom riešení a riešenie uložíme. Ak naopak takéhoto reprezentanta nenájdeme, uložíme aktuálne riešenie do tabuľky `bad_formalizations` a ako cudzí kľúč k riešeniu nám poslúži jeho id.

## 5.3 Pridávanie nápovedí

Aby mohli učitelia pridávať nápovede k chybným formalizáciám, ako sme navrhli v sekcii 4.2.2, bolo potrebné najskôr zobraziť tieto chybné formalizácie k jednotlivým tvrdeniam. Na strane klienta sme preto vytvorili nový komponent `BadExercise.js` zobrazujúci zoznam všetkých cvičení, pričom pri každom cvičení sa zobrazuje počet reprezentantov chybných formalizácií tvrdení v danom cvičení a počet študentov, ktorí cvičenie riešili. Po vybratí niektorého cvičenia sa pomocou nového kom-

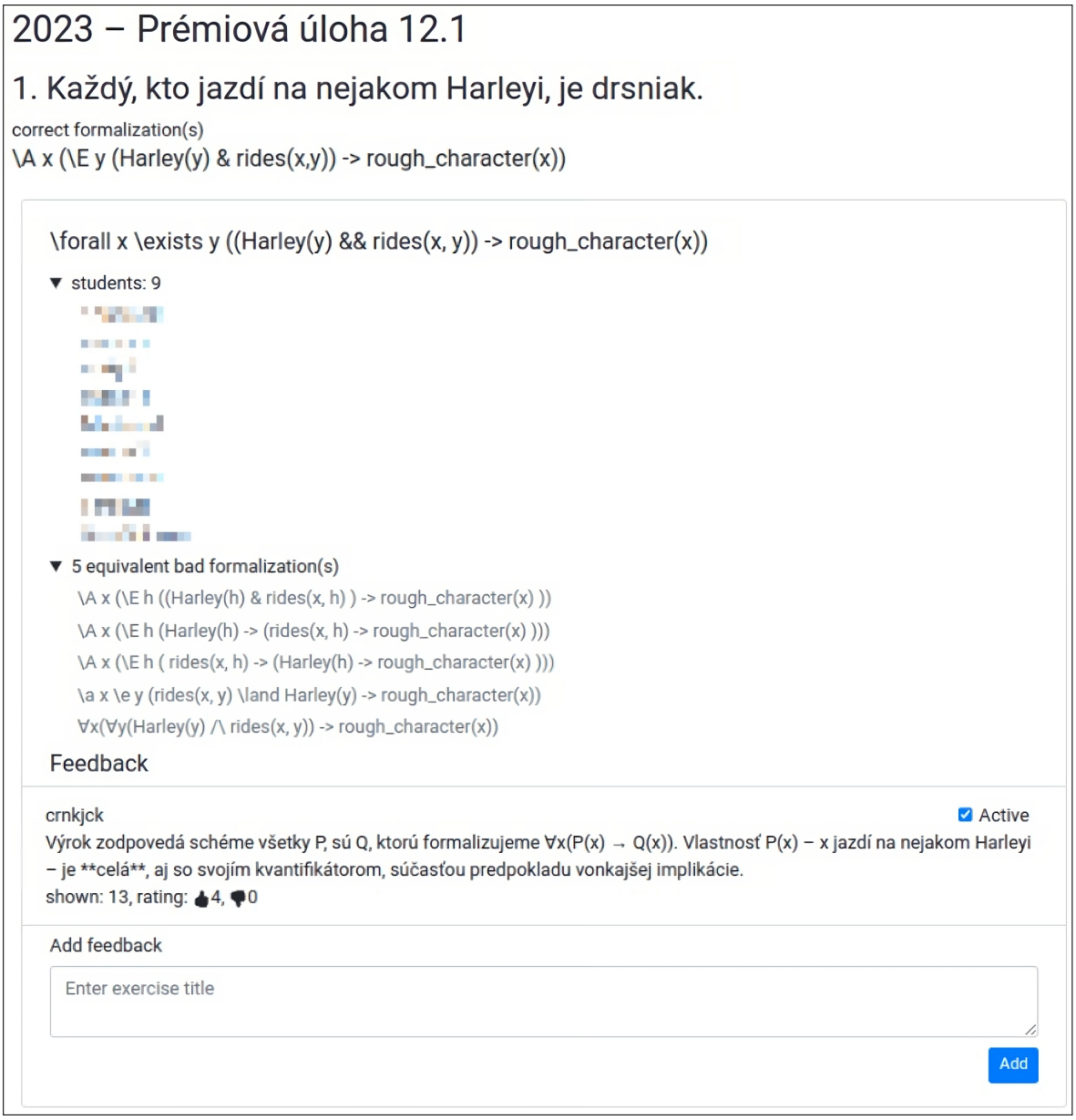
ponentu `BadPropositionsToExercise.js` zobrazia tvrdenia na sformalizovanie daného cvičenia, taktiež spolu s počtom reprezentantov chybných formalizácií daného tvrdenia a počtom študentov, ktorí riešili dané tvrdenie. Následne, po zvolení niektorého tvrdenia, sa zobrazia karty obsahujúce všetkých reprezentantov chybných formalizácií tohto tvrdenia a pre zjednodušenie sa zobrazí aj správne riešenie formalizácie. Každá karta obsahuje reprezentanta chybnej formalizácie, zoznam študentov, ktorí spravili ekvivalentnú chybnú formalizáciu a zoznam všetkých ekvivalentných chybných riešení. Navyše sa učiteľovi, kliknutím na meno študenta, zobrazí postup tohto študenta v riešení cvičenia. Na vykresľovanie týchto údajov sme vytvorili nový komponent `BadFormalizationsToProposition.js`, ktorý obsahuje ďalšie novo-vytvorené komponenty. Na manažovanie stavu týchto komponentov sme taktiež vytvorili dva nové súbory `badFormalizationsSlice.js` a `feedbacksSlice.js` za využitia RTK Query.

V každej karte reprezentanta chybnej formalizácie, sa zobrazuje aj zoznam existujúcich nápovedí s možnosťou pridať novú nápoveď. Pri každej už pridanej nápovedi sa zobrazuje jej autor, počet študentov, ktorým sa nápoveď zobrazila a počet študentov, ktorí ju hodnotili, buď palcom hore, alebo dole. Ak kurzorom myši nadídeme nad tieto počty, zobrazí sa navyše zoznam študentov, ktorým sa nápoveď zobrazila, ktorí ju ohodnotili palcom hore, a ktorí ju ohodnotili palcom dole. Takisto má učiteľ možnosť nastaviť, či bude nápoveď aktívna, a teda sa bude zobrazovať študentom, alebo bude skrytá. Pre pridanie novej nápovede, učiteľ napíše text nápovede a po stlačení tlačidla sa nápoveď pridá do zoznamu nápovedí, pričom je vopred nastavená ako aktívna. Ukážku zobrazenia zoskupených chybných formalizácií spolu s nápoveďami môžeme vidieť na obrázku 5.1.

Na ukladanie nápovedí sme využili tabuľku `feedbacks`, ktorú sme vytvorili pomocou súboru `db/migrations/002-feedback-to-solution.sql`. Každý záznam obsahuje referenciu na id reprezentanta chybnej formalizácie, vďaka čomu vieme správne priraďovať nápovede. Na strane servera sme navyše pridali modul `api/feedbacks.js`, ktorý obsluhuje cesty s prefixom `api/feedbacks` a spracováva požiadavky zo strany klienta, týkajúce sa zobrazovania, pridávania či editovania nápovedí. Tento modul využíva databázové funkcie v ďalšom pridanom module `db/feedbacks.js`, ktorý komunikuje s databázou.

## 5.4 Zobrazovanie a hodnotenie nápovedí

Ak študent chybne vyrieši tvrdenie a jeho formalizácia je ekvivalentná niektorej formalizácii, ktorá už má pridané nejaké nápovede, bude si môcť tieto nápovede zobraziť. Táto kontrola prebieha po odoslaní riešenia. Ak študentovo riešenie nie je správne, priradí sa mu reprezentant chybnej formalizácie, podľa postupu v sekcii 5.2. Následne




Obr. 5.1: Chybné riešenia a nápovede z pohľadu učiteľa

sa získajú všetky aktívne nápovede k danému reprezentantovi chybnej formalizácie. Ak takéto nápovede existujú, študentovi sa zobrazí tlačidlo, pomocou ktorého si ich môže postupne zobrazovať, ako sme navrhli v sekcii 4.2.3. Pri každej nápovedi sa zobrazí ikona palca hore a palca dole, pomocou ktorých môže študent ohodnotiť danú nápoveď.

Pomocou skriptu `db/migrations/001-add-bad-formalizations.js` sme vytvorili tabuľku `feedback_to_solution`, ktorá slúži na evidovanie hodnotenia nápovede k danému riešeniu. Akonáhle si študent vyžiada zobraziť nápoveď, vyrobíme záznam v tejto tabuľke, pričom bude obsahovať referenciu na id jeho riešenia a id nápovede, ktorá sa mu zobrazila. Ak študent ohodnotí danú nápoveď, či už palcom hore, alebo palcom dole, príslušne zmeníme atribút `rating` daného záznamu. V momente ako štu-

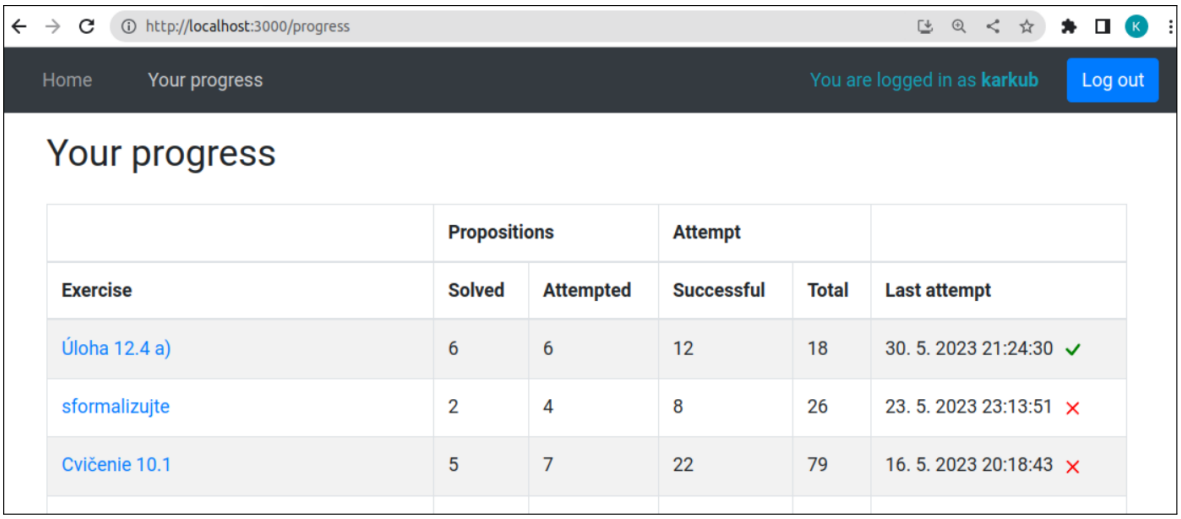
dent odošle novú formalizáciu daného tvrdenia, všetky zobrazené nápovede zmiznú a vyššie opísaný proces sa zopakuje.


Obr. 5.2: Zobrazenie a hodnotenie nápovede k riešeniu z pohľadu študenta

Ako vidíme na obrázku 5.2, v prípade nesprávneho riešenia, ku ktorému existujú nejaké nápovede, sa používateľovi zobrazí tlačidlo `Hint`. Po kliknutí naň sa používateľovi postupne zobrazia nápovede. Ak si už zobrazil všetky nápovede toto tlačidlo zmizne, ako môžeme vidieť pri druhom tvrdení na obrázku 5.2. Pri každej nápovedi, sa najskôr zobrazia nevyplnené ikony palca hore a palca dole. Študent môže niektorú nápoveď ohodnotiť kliknutím na niektorú ikonu, ktorá sa následne vyplní.

## 5.5 Zobrazovanie postupu riešenia cvičení študentovi

Študentom sme umožnili zobrazovať ich postup v riešení cvičení, podobne ako to bolo umožnené aj učiteľom, podľa návrhu v sekcii 4.2.4. Študentovi sa po kliknutí na `Your progress` v menu zobrazí zoznam cvičení, ktoré riešil, pričom pri každom cvičení sa zobrazuje počet tvrdení, ktoré riešil, počet tvrdení, ktoré úspešne vyriešil a počet celkových a úspešných pokusov. Takisto sa zobrazuje aj dátum posledného pokusu s informáciou, či bol úspešný, alebo nie. Toto zobrazenie môžeme vidieť na obrázku 5.3. Po vybratí niektorého cvičenia sa zobrazia tvrdenia, ktoré študent riešil spolu so všetkými riešeniami. Pri každom riešení sa taktiež zobrazuje dátum riešenia a informácia, či bolo riešenie správne, alebo nie. Takisto sa k riešeniu zobrazuje aj zoznam nápovedí, ktoré si zobrazil, spolu s ikonou palca hore, alebo dole, ak nápoveď ohodnotil. Toto zobrazenie vidí študent podobne ako učiteľ na obrázku 5.4, s rozdielom absencie jednotlivých odkazov na nápovediach.

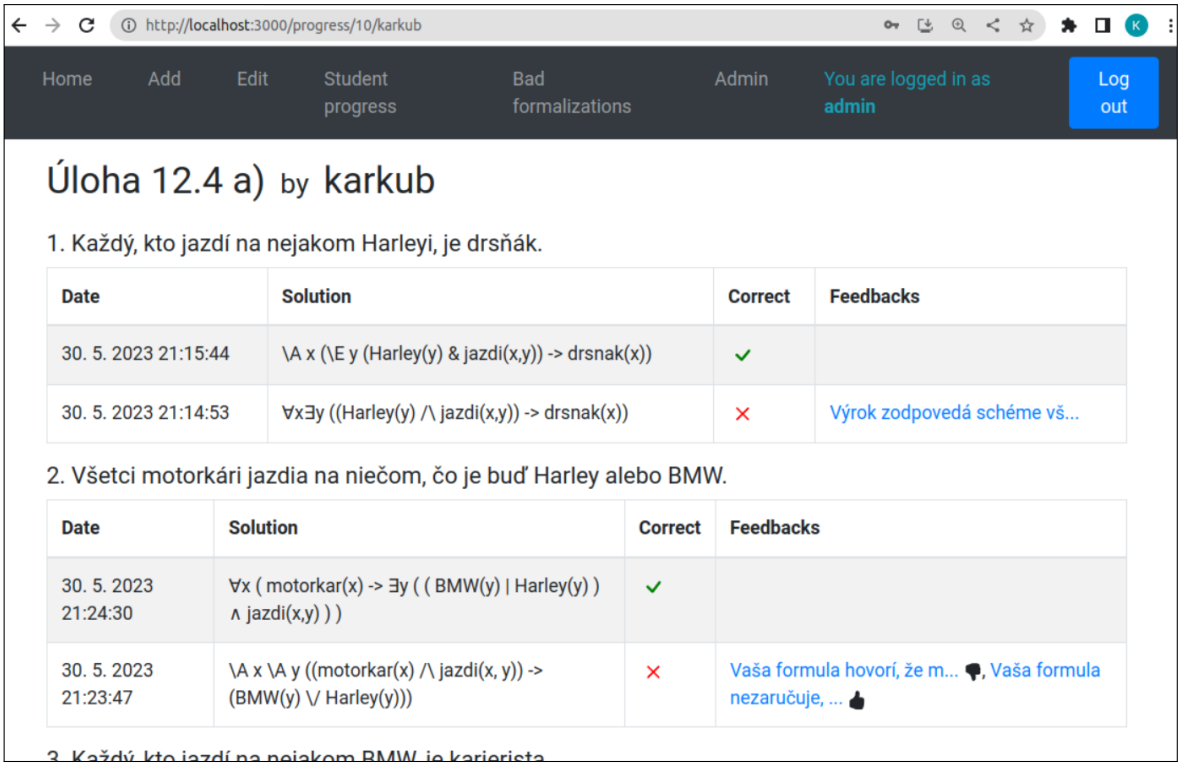



Obr. 5.3: Štatistika riešení cvičení študenta

## 5.6 Zobrazovanie postupu študentov v riešení cvičení

V tejto časti aplikácie, zobrazovania postupu študentov v riešení cvičení, sme urobili iba malé zmeny oproti pôvodnej implementácii. Učiteľovi sa najskôr zobrazí zoznam všetkých cvičení, spolu s počtom študentov, ktorí danú úlohu riešili, pričom sa do tohto počtu už nepočítajú učitelia. Po zvolení niektorého cvičenia sa štandardne zobrazí zoznam používateľov, ktorí danú úlohu riešili. Ak je niektorý používateľ zároveň aj administrátor, zobrazí sa k jeho menu navyše aj ikona označujúca túto skutočnosť. Takisto sa pri každom používateľovi zobrazuje aj štatistika, ktorá sa ale v minulosti vypočítavala nesprávne, a preto sme ju opravili úpravou databázových volaní.

Výraznejšou novinkou je, že pri zobrazení postupu riešenia nejakého cvičenia konkrétneho študenta, sa navyše zobrazuje ktoré nápovede si študent zobrazil a pri ktorom riešení, spolu s ikonou palca hore, alebo dole, ak študent nápoveď ohodnotil. Túto funkciu môžeme vidieť na obrázku 5.4. Aj takouto formou vie učiteľ sledovať, či rôzne nápovede viedli k úspešnému vyriešeniu tvrdenia. Navyše sa učiteľovi, po kliknutí na nejakú nápoveď, zobrazí daná nápoveď pridaná ku niektorému reprezentantovi danej chybnej formalizácie, kde môže vidieť bližšie detaily, ako sme opísali v sekcii 5.3.




Obr. 5.4: Postup študenta v riešení cvičení, z pohľadu učiteľa

# Kapitola 6

# Testovanie

V tejto kapitole si opíšeme ako prebiehalo testovanie aplikácie a uvedieme si výsledky testovania. Aplikácia bola nasadená do prevádzky a je prístupná na adrese `https://fmfi-uk-1-ain-412.github.io/formalization-checker/`.

## 6.1 Cieľ testovania

Hlavným cieľom testovania bolo vyhodnotiť užitočnosť nápovedí pri riešení cvičenia. Ďalším cieľom bolo otestovať, či sú všetky funkcie implementované správne a prípadne rôzne chyby opraviť, a taktiež zistiť rôzne nedostatky, ktoré by mohli byť ďalej implementované v budúcnosti.

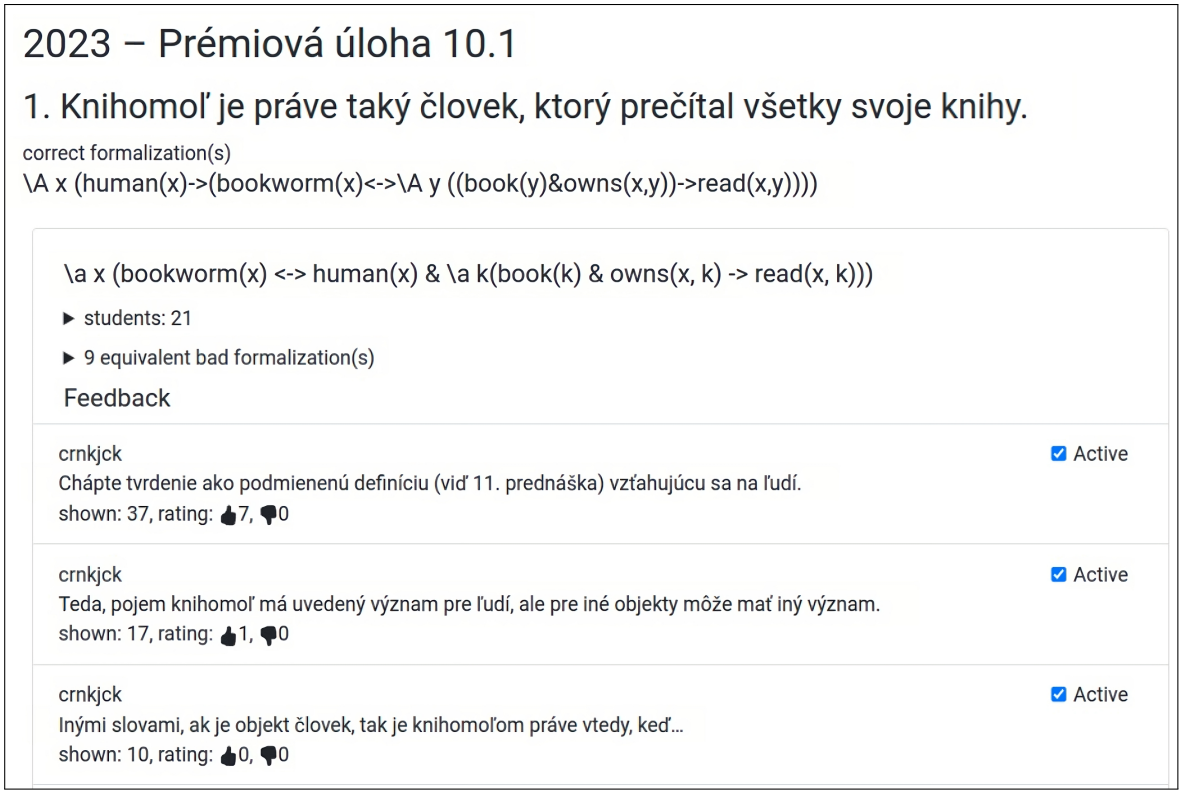
## 6.2 Priebeh testovania

Aplikácia bola testovaná študentmi druhého, prípadne tretieho ročníka na predmete Logika pre informatikov.

Nápovede boli pridané k dvom rôznym cvičeniam, v ktorých by potenciálne, na základe analýzy dát z minulého roka, mohli študenti robiť viac chýb. Učitelia vopred pridali rôzne chybné riešenia viacerých tvrdení v týchto dvoch cvičeniach, ktoré spravili aspoň dvaja študenti minulý rok. Vďaka tomu vyrobili rôznych reprezentantov chybných formalizácií, ku ktorým následne pridali nápovede. Tohtoročným študentom, ktorí riešili dané úlohy, sa teda hneď mohli zobrazovať tieto nápovede. Užitočnosť nápovedí vieme vyhodnotiť na základe hodnotenia rôznych nápovedí študentami, či už palcom hore, alebo palcom dole. Takisto môžeme zanalyzovať postup niektorého študenta v riešení nejakého tvrdenia, ak sa mu zobrazili nejaké nápovede.

## 6.3 Výsledok testovania

Ako môžeme vidieť na obrázku 6.1, pri danom reprezentantovi chybnej formalizácie tvrdenia *Knihomoľ je práve taký človek, ktorý prečítal všetky svoje knihy* boli pridané 3 nápovede. 21 študentov spravilo nejakú chybnú formalizáciu ekvivalentnú tomuto reprezentantovi, pričom prvá nápoveď sa zobrazila až 37-krát, a pozitívne, palcom hore, ju ohodnotilo 7 študentov. Na základe tohto môžeme usúdiť, že študenti mali záujem zobrazovať si nápovede k tejto formalizácii. Ďalšie dve nápovede sa však zobrazovali už menej. Môže to byť v dôvodu, že študenti nemali potrebu zobraziť si ďalšiu nápoveď, pretože im pomohla tá prvá, alebo o to naopak nemali záujem.




Obr. 6.1: Analýza užitočnosti nápovedí na základe zobrazovania a hodnotenia nápovedí

Ďalším spôsobom ako môžeme vyhodnotiť užitočnosť nápovedí je, že sa pozrieme sa postup študenta v riešení tvrdenia, ak si zobrazil nejaké nápovede. Na obrázkoch 6.2 a 6.3 vidíme, že študenti po zobrazení nápovedí správne vyriešili dané tvrdenie, z čoho môžeme usúdiť, že dané im nápovede asi pomohli. Takýchto prípadov bolo viac, no pre stručnosť neuvádzame ďalšie obrázky. Je ich však možné vidieť v aplikácii, v zobrazení postupu študentov v riešení úloh. Existujú ale aj niektorí študenti, ktorým sa aj po zobrazení nápovede nepodarilo správne vyriešiť formalizáciu, ako môžeme vidieť na obrázku 6.4.

Z hľadiska učiteľa bolo prácne pripravovať nápovede k chybným formalizáciám, nakoľko chybných riešení bolo veľa. Takisto sa viaceré chyby, ako napríklad zámena ekvi-

2023 – Prémiová úloha 10.1 by [redacted]

1. Knihomoľ je práve taký človek, ktorý prečítal všetky svoje knihy.

| Date | Solution | Correct | Feedbacks |
|---|---|---|---|
| 3. 5. 2023 15:44:59 | \A x (bookworm(x) <-> \A y (owns(x,y) & book(y) & human(x) -> read(x, y))) | ✗ | Chápte tvrdenie ako podmi... , Vaša formula hovorí, že k... |
| 3. 5. 2023 15:44:06 | \A x (bookworm(x) <-> \A y (book(y) & human(x) & owns(x,y) & read(x, y))) | ✗ | Podľa vašej formuly má by... |
| 3. 5. 2023 15:43:43 | \A x (bookworm(x) <-> \E y (book(y) & human(x) & owns(x,y) & read(x, y))) | ✗ | |
| 3. 5. 2023 15:42:10 | \A x (-bookworm(x) <-> \E y (book(y) & human(x) & owns(x,y) & -read(x, y))) | ✗ | Vaša formula hovorí, že k... |

Obr. 6.2: Analýza užitočnosti nápovedí na základe postupu študenta v riešení

2023 – Prémiová úloha 10.1 by [redacted]

1. Knihomoľ je práve taký človek, ktorý prečítal všetky svoje knihy.

| Date | Solution | Correct | Feedbacks |
|---|---|---|---|
| 10. 5. 2023 0:12:53 | \forall x (human(x) \limpl (bookworm(x) \lequiv \forall y ((book(y) \land owns(x,y)) \limpl read(x,y)))) | ✓ | |
| 10. 5. 2023 0:12:26 | \forall x (bookworm(x) \limpl (human(x) \lequiv \forall y ((book(y) \land owns(x,y)) \limpl read(x,y)))) | ✗ | Vaša formula nedefinuje, ... |
| 10. 5. 2023 0:09:45 | \forall x (bookworm(x) \lequiv (human(x) \land \forall y ((book(y) \land owns(x,y)) \limpl read(x,y)))) | ✗ | Chápte tvrdenie ako podmi... |

Obr. 6.3: Analýza užitočnosti nápovedí na základe postupu študenta v riešení

valencie s implikáciou, či spojenie kvantifikátorov s nesprávnymi spojkami, prekrývali a bolo potrebné jednotlivé nápovede kopírovať k viacerým reprezentantom chybných formalizácií.

2023 – Prémiová úloha 10.1 by [redacted]

1. Knihomoľ je práve taký človek, ktorý prečítal všetky svoje knihy.

| Date | Solution | Correct | Feedbacks |
|---|---|---|---|
| 25. 5. 2023 1:44:17 | ∀x ( (bookworm(x) & human(x)) <->( human(x) & ∀y ( (book(y) & owns(x, y)) -> read(x, y)))  ) | ✓ | |
| 25. 5. 2023 1:43:51 | ∀x ( (bookworm(x) & human(x)) <->∀y ( (book(y) & owns(x, y) & human(x)) -> read(x, y))) | ✗ | |
| 25. 5. 2023 1:43:27 | ∀x ( (bookworm(x) & human(x)) <->∀y ( (book(y) & owns(x, y)) -> read(x, y))) | ✗ | Vaša formula hovorí, že k... |
| 25. 5. 2023 1:43:03 | ∀x ( (bookworm(x) & human(x)) ->∀y ( (book(y) & owns(x, y)) -> read(x, y))) | ✗ | Vaša formula požaduje, ab... |
| 25. 5. 2023 1:42:52 | ∀x ( (bookworm(x) & human(x)) <->∀y ( (book(y) & owns(x, y)) -> read(x, y))) | ✗ | Vaša formula hovorí, že k... |
| 25. 5. 2023 1:42:18 | ∀x ( bookworm(x) <-> ( human(x) & ∀y ( (book(y) & owns(x, y)) -> read(x, y)))  ) | ✗ | Teda, pojem knihomoľ má u... , Chápte tvrdenie ako podmi... |

Obr. 6.4: Analýza užitočnosti nápovedí na základe postupu študenta v riešení

# Záver

Do existujúcej aplikácie na kontrolu formalizačných cvičení sme implementovali možnosť pridávať nápovede k chybným formalizáciám, ktoré zoskupujeme na základe ekvivalencie, možnosť zobrazovať a hodnotiť tieto nápovede pri danom chybnom riešení. Pred samotnou implementáciou týchto funkcií sme najskôr refaktorovali kód z predošlých dvoch implementácií, aby bol logickejšie usporiadaný, modulárnejší a jednoduchší na ďalšiu úpravu.

Jednotlivé chybné riešenia sme najskôr zoskupili na základe ekvivalencie. Učiteľ si môže tieto zoskupené chybné formalizácie zobraziť a každému reprezentantovi chybnej formalizácie môže pridať niekoľko nápovedí. Učiteľovi sa takisto zobrazuje počet zobrazení danej nápovede, spolu s reakciami, ako ju študenti hodnotia. Na základe toho môžu učitelia negatívne hodnotené nápovede deaktivovať, prípadne pridať ďalšie doplňujúce nápovede. Zobrazenie nápovedí s ich hodnotením k jednotlivým riešeniam môže učiteľ sledovať taktiež v zobrazení postupu študentov v riešení úloh, a tak analyzovať, či rôzne nápovede skutočne napomohli študentovi k správnemu vyriešeniu tvrdenia.

Ak študent urobí chybné riešenie, ekvivalentné niektorému, ktoré už má priradenú nejakú nápoveď, prípadne viaceré nápovede, študent si ich môže postupne zobraziť. Každú nápoveď môže navyše ohodnotiť podľa toho, či ju považuje za užitočnú, alebo nie. Študent môže taktiež sledovať svoj postup v riešení úloh, kde sa mu tiež zobrazujú, ktoré nápovede si zobrazil pri akom riešení.

Naša aplikácia je nasadená na adrese `https://fmfi-uk-1-ain-412.github.io/formalization-checker/` a využíva sa na predmete Logika pre informatikov. Na základe testovania študentmi tohto predmetu, sa zdá, že nápovede boli užitočné. Viacerým študentom sa po zobrazení nejakej nápovede podarilo správne vyriešiť dané tvrdenie. Viaceré nápovede majú tiež vyšší počet zobrazení, čo môže ukazovať, že študenti majú záujem si zobrazovať nápovede.

Pridávanie nápovedí je však veľmi prácne, nakoľko chybných formalizácií je veľa a niektoré chyby sa opakujú pri viacerých reprezentantoch chybných formalizácií. Prípadným ďalším rozšírením tejto aplikácie by preto mohla byť implementácia automatického generovania nápovedí na základe syntaktických vlastností formúl. Napríklad, v prípade, že správna formalizácia obsahuje ekvivalenciu, ale rôzne chybné riešenia ekvivalenciu neobsahujú, automaticky by sa im vygenerovala nápoveď, popisujúca túto

skutočnosť.

# Literatúra


[1] Samantha Gombárová. Automatizácia kontroly formalizačných cvičení v logike prvého rádu. Bakalárska práca, Univerzita Komenského v Bratislave, Bratislava, 2021.

[2] Nikola Kulíková. Automatická spätná väzba na riešenia formalizačných cvičení. Bakalárska práca, Univerzita Komenského v Bratislave, Bratislava, 2022.

[3] Ján Kľuka, Ján Mazák, and Jozef Šiška. Logika pre informatikov a Úvod do matematickej logiky. Poznámky z prednášok, Letný semester 2022/2023. [Dostupné 12.5.2023] `https://fmfi-uk-1-ain-412.github.io/lpi/prednasky/poznamky-z-prednasok.pdf`.

[4] Vladimír Kvasnička and Jiří Pospíchal. *Matematická logika*. Vydavateľstvo STU, Bratislava, 2006.

[5] StrongLoop, IBM, and other expressjs.com contributors. *Express*. [Dostupné 8.5.2023] `https://expressjs.com/`.

[6] Meta Open Source. *React*. [Dostupné 8.5.2023] `https://react.dev/`.

[7] Remix Software, Inc. *React Router*. [Dostupné 20.5.2023] `https://reactrouter.com/en/main`.

[8] Dan Abramov and the Redux documentation authors. *Redux*. [Dostupné 8.5.2023] `https://redux.js.org/`.

[9] Dan Abramov and the Redux documentation authors. *RTK Query*. [Dostupné 20.5.2023] `https://redux-toolkit.js.org/rtk-query/overview`.

[10] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044*, CAV 2013, page 1–35, Berlin, Heidelberg, 2013. Springer-Verlag.

# Príloha A: Elektronická príloha

Elektronická príloha obsahuje zdrojový kód aplikácie, po zmenách opísaných v tejto práci. Zdrojový kód je rozdelený do priečinkov:

- `backend` – zdrojový kód aplikácie na strane servera
- `frontend` – zdrojový kód aplikácie na strane klienta.

Zdrojové kódy sa nachádzajú aj online na adresách:

- `https://github.com/FMFI-UK-1-AIN-412/formalization-checker-backend`
- `https://github.com/FMFI-UK-1-AIN-412/formalization-checker`.

Aplikácia je nasadená na adrese `https://fmfi-uk-1-ain-412.github.io/formalization-checker/`.